



mastering python



Title: Mastering Python – Basics to Machine Learning

Author: Dr Gunnar Mallon

© 2024 by Dr Gunnar Mallon

All rights reserved. No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the author, except in the case of brief quotations embodied in critical reviews and certain other non-commercial uses permitted by copyright law.

First Edition, 2024

Acknowledgements:

Thanks to Giulia, Dolce, and Buddy

Companion Resources:

This book is accompanied by a video series on YouTube at <https://www.youtube.com/@CodewithGunnar/>. These videos provide visual demonstrations of key concepts and coding examples. Readers are encouraged to use these resources in conjunction with the book for a more comprehensive learning experience.

Additionally, code samples, datasets, and supplementary materials referenced in this book can be found at: <https://codewithgunnar.com>

How to Use This Book

This book is structured into chapters, each covering a specific aspect of Python programming. It is recommended that you follow the chapters in sequence, but you may also refer to specific sections as needed. Each chapter includes practical examples and exercises designed to reinforce the concepts covered.

Companion Resources

Additional resources, including code examples and exercises, can be found at <https://codewithgunnar.com>. These resources are intended to complement the material presented in this book.

Contact Information

For questions, feedback, or further discussion, please contact me at g.mallon@rug.nl or join us in the course Discord channel: <https://discord.gg/WyA9ypqZ6p>.



Dear Student and future Pythonista,

Welcome to "Mastering Python: From Basics to Machine Learning." This book represents the culmination of years of teaching, research, and practical experience in the exciting field of geospatial data science.

As an Assistant Professor at the University of Groningen's Faculty of Spatial Sciences, I have had the privilege of guiding students through the intricate world of geospatial analysis and Python programming. This book originated from a course I developed for the minor in geospatial data science, and it has since evolved into a comprehensive resource that extends far beyond the classroom.

The journey of creating this book has been both challenging and rewarding. What began as course material has grown into a full-fledged guide that not only covers the fundamentals of Python programming for geospatial applications but also delves into advanced topics such as machine learning in the spatial context. This expansion reflects the rapidly evolving nature of our field and the increasing demand for professionals who can leverage these powerful tools.

In parallel with this book, I have developed a video series available on my YouTube channel, "Code with Gunnar." These videos complement the written content, offering visual demonstrations of key concepts and practical coding examples. I encourage readers to utilize both resources for a more immersive learning experience.

This book is designed to cater to a wide audience – from university students embarking on their geospatial journey to professionals seeking to enhance their skills in Python and machine learning within a spatial context. The content is structured to provide a gradual progression from basic Python concepts to advanced geospatial analysis techniques and machine learning applications.

Key features of this book include:

- Comprehensive coverage of Python basics tailored for geospatial applications
- In-depth exploration of geospatial data handling and analysis using Python
- Introduction to machine learning concepts and their application in spatial analysis
- Practical examples and case studies drawn from real-world scenarios
- Hands-on exercises to reinforce learning and develop practical skills
- Links to supplementary video content for enhanced understanding

My goal in writing this book is not just to impart knowledge but to inspire curiosity and innovation in the field of geospatial data science. The intersection of geography, data science, and machine learning offers immense potential for solving complex spatial problems and gaining new insights into our world.

I sincerely hope that this book serves as a valuable resource in your learning journey and professional development. Whether you're a student, researcher, or industry professional, I trust you'll find the content both informative and practical.

Thank you for choosing this book. I look forward to being a part of your exploration into the fascinating world of geospatial data science with Python.

Happy coding!

Dr. Gunnar Mallon, PhD, FRGS, CEnv
Assistant Professor, Faculty of Spatial Sciences, University of Groningen
Creator, Code with Gunnar

INTRODUCTION.....	1	IF-ELIF-ELSE CHAINS	18
INSTALLATION AND SETUP	2	NESTED CONDITIONALS.....	18
INSTALLING VSCODE	2	TERNARY OPERATORS	18
SETTING UP VSCODE.....	2	COMPLEX CONDITIONS	19
PYTHON SYNTAX	3	SHORT-CIRCUIT EVALUATION.....	19
PRINT() OPTIONS	4	THE MATCH STATEMENT	19
THE HELP() FUNCTION	4	LOOPS.....	20
COMMENTS IN PYTHON	4	THE 'FOR' LOOP.....	20
LINE CONTINUATION.....	5	THE 'WHILE' LOOP.....	21
CASE SENSITIVITY	5	LOOP CONTROL STATEMENTS	21
VARIABLES AND DATA TYPES	5	THE ENUMERATE() FUNCTION	22
CREATING VARIABLES	6	INFINITE ITERATORS (ADVANCED)	22
VARIABLE NAMING RULES AND CONVENTIONS	6	COMMON LOOP PATTERNS.....	23
NAMING BEST PRACTICES.....	6	NESTED LOOPS.....	24
BASIC DATA TYPES FOR VARIABLES	6	THE ITER() AND NEXT() FUNCTIONS	24
VARIABLE TYPES	7	LISTS	25
VARIABLE REASSIGNMENT	7	CREATING LISTS	25
MULTIPLE ASSIGNMENT	7	LIST ELEMENTS.....	25
CONSTANTS	7	ACCESSING LIST ELEMENTS	25
PRINT() FUNCTION WITH VARIABLES	7	LIST SLICING	26
THE NONE VALUE.....	8	MODIFYING LISTS	26
TYPE CONVERSION	8	LIST METHODS.....	26
CHECKING VARIABLE EXISTENCE.....	8	LIST OPERATIONS.....	28
VARIABLES IN SIMPLE EXPRESSIONS	8	LIST LENGTH	28
UNDERSCORE VARIABLE	8	CHECKING MEMBERSHIP	28
STRINGS	9	LIST COMPREHENSIONS.....	28
CREATING STRINGS.....	9	COPYING LISTS	29
STRING INDEXING	9	LIST UNPACKING	30
STRING SLICING	10	THE NONE OBJECT IN LISTS.....	30
STRING CONCATENATION.....	10	REMOVING DUPLICATES FROM A LIST	30
STRING METHODS	11	LISTS IN STRING FORMATTING	30
STRING FORMATTING WITH F-STRINGS	12	COMPARING LISTS.....	30
ESCAPE CHARACTERS	12	THE DEL STATEMENT WITH LISTS	31
STRING IMMUTABILITY	13	TUPLES	31
THE 'IN' OPERATOR WITH STRINGS.....	13	CREATING TUPLES	31
SPLITTING AND JOINING STRINGS	13	TUPLE PACKING AND UNPACKING.....	32
CHECKING STRING PROPERTIES.....	13	ACCESSING TUPLE ELEMENTS	32
REPLACING PARTS OF STRINGS	14	TUPLE SLICING	32
STRING ALIGNMENT	14	IMMUTABILITY OF TUPLES.....	33
ZFILL() FOR PADDING	14	TUPLE METHODS	33
COUNTING OCCURRENCES	15	TUPLE OPERATIONS.....	33
WORKING WITH NUMERIC STRINGS	15	ITERATING OVER TUPLES	34
CONDITIONALS	16	TUPLES VS. LISTS.....	34
INTRODUCTION TO CONDITIONALS	16	CONVERTING BETWEEN TUPLES AND LISTS.....	34
THE IF STATEMENT	16	TUPLE COMPREHENSIONS.....	35
IF-ELSE STATEMENTS.....	17	THE 'ZIP()' FUNCTION.....	35
		TUPLES IN LIST COMPREHENSIONS	35
		SETS.....	36

CREATING SETS	36	FILE OPERATIONS	57
SET PROPERTIES	37	FILE MODES	58
ADDING ELEMENTS TO SETS	37	READING FILES	58
REMOVING ELEMENTS FROM SETS	37	WRITING FILES	58
SET OPERATIONS	38	FILE POSITIONS	59
SET METHODS FOR OPERATIONS	38	WORKING WITH CSV FILES	59
SUBSET AND SUPERSET	39	WORKING WITH JSON FILES	59
CHECKING MEMBERSHIP	40	FILE SYSTEM OPERATIONS (OPTIONAL)	60
SETS IN BOOLEAN CONTEXT	40	WORKING WITH PATHS	61
DISJOINT SETS	40	FILE ENCODING	61
SETS IN LIST COMPREHENSIONS	40	FILE COMPRESSION	61
BEST PRACTICES FOR USING SETS	41	ERROR HANDLING	62
CREATING DICTIONARIES	41	EXCEPTION HANDLING: TRY-CATCH BLOCK	62
ACCESSING DICTIONARY ELEMENTS	42	ERROR HANDLING BEST PRACTICES	63
MODIFYING DICTIONARIES	42	ERROR MESSAGES AND USER COMMUNICATION	63
REMOVING ITEMS FROM DICTIONARIES	42	DEBUGGING TECHNIQUES	64
DICTIONARY METHODS	43	TESTING ERROR HANDLING CODE	65
DICTIONARY COMPREHENSIONS	43	OBJECT-ORIENTED PROGRAMMING	65
DICTIONARY AND SET VIEW OBJECTS	44	CLASSES	66
MERGING DICTIONARIES	44	OBJECTS	66
DICTIONARIES VS. LISTS OF TUPLES	44	ATTRIBUTES (PROPERTIES)	67
SORTING DICTIONARIES	44	METHODS	67
INVERTING A DICTIONARY	45	CONSTRUCTORS	68
DICTIONARY AND JSON	45	DESTRUCTORS IN PYTHON	68
HANDLING MISSING KEYS	45	ENCAPSULATION	69
REAL-WORLD SCENARIOS	45	ABSTRACTION	69
FUNCTIONS	46	BASIC CLASS RELATIONSHIPS	69
DEFINING FUNCTIONS	46	‘SELF’ KEYWORD (ADVANCED)	70
FUNCTION PARAMETERS AND ARGUMENTS	47	OBJECT COMPARISON	70
RETURN STATEMENTS	48	OBJECT SERIALIZATION	71
DOCSTRINGS	48	BEST PRACTICES	71
SCOPE AND NAMESPACE	49	COMMON DESIGN PATTERNS	72
FUNCTIONS AS OBJECTS	49	INHERITANCE AND POLYMORPHISM	72
LAMBDA FUNCTIONS	50	TYPES OF INHERITANCE	73
RECURSIVE FUNCTIONS	50	METHOD INHERITANCE	74
HIGHER-ORDER FUNCTIONS	51	CONSTRUCTOR INHERITANCE	75
DECORATORS	51	ABSTRACT CLASSES	75
FUNCTION ANNOTATIONS (OPTIONAL)	52	COMPOSITION VS INHERITANCE	75
GENERATOR FUNCTIONS	52	POLYMORPHISM BASICS	76
CALLBACK FUNCTIONS	52	METHOD OVERRIDING (IN-DEPTH)	76
MODULES AND PACKAGES	53	MIXINS AND TRAITS	77
IMPORTING MODULES	54	THE TEMPLATE METHOD PATTERN	77
BUILT-IN MODULES	55	THE STRATEGY PATTERN	77
COMMONLY USED BUILT-IN MODULES	55	COMMON PITFALLS AND BEST PRACTICES	78
INTRODUCTION TO PACKAGES	55	NUMPY	78
VIRTUAL ENVIRONMENTS	56	INSTALLING AND IMPORTING NUMPY	79
PYTHON PACKAGE INDEX (PYPi)	56	NUMPY ARRAYS	79
FILES	57		

CREATING NUMPY ARRAYS	79	PANDAS	117
APPENDING AND ADDING ELEMENTS	81	CORE DATA STRUCTURES	117
ARRAY ATTRIBUTES	81	CREATING SERIES OBJECTS	117
ARRAY INDEXING AND SLICING	82	[CREATING DATAFRAME OBJECTS]	118
ARRAY RESHAPING	83	[BASIC ATTRIBUTES AND METHODS OF SERIES AND	
ARRAY CONCATENATION AND SPLITTING	84	DATAFRAMES]	120
DATA TYPES IN NUMPY	85	[ACCESSING DATA IN SERIES]	122
TYPE CONVERSION	86	[ACCESSING DATA IN DATAFRAMES]	124
BROADCASTING	87	[BASIC OPERATIONS ON SERIES AND DATAFRAMES]	125
BASIC ARRAY OPERATIONS	88	[HANDLING MISSING DATA]	126
UNIVERSAL FUNCTIONS (UFUNCS)	89	[INDEX OBJECTS]	128
ARRAY INPUT/OUTPUT	89	[REINDEXING]	129
BASIC LINEAR ALGEBRA OPERATIONS	90	[SORTING]	130
INTRODUCTION TO STRUCTURED ARRAYS	91	[BASIC PLOTTING WITH PANDAS]	132
BASIC ARRAY MANIPULATION	91	[BASIC STATISTICAL METHODS]	133
NUMPY ECOSYSTEM	92	[BASIC DATA MANIPULATION]	134
ARRAY OPERATIONS	93	[INTRODUCTION TO PANDAS' ECOSYSTEM]	135
INDEXING TECHNIQUES (REVISITED)	94	[BEST PRACTICES AND CONVENTIONS]	136
SET OPERATIONS	95	[RESOURCES FOR FURTHER LEARNING]	137
SORTING ARRAYS	96	DATAFRAME OPERATIONS	139
ARRAY COMPARISON OPERATIONS	96	REVIEW OF DATAFRAME BASICS	139
LOGICAL OPERATIONS ON ARRAYS	97	SELECTING AND FILTERING DATA	140
ARRAY ITERATION	97	ADDING AND REMOVING COLUMNS	141
ARRAY MANIPULATION USING UFUNCS	99	ADDING AND REMOVING ROWS	142
RECORD ARRAYS	99	HANDLING MISSING DATA (ADVANCED)	143
ARRAY PADDING	100	SORTING AND ORDERING	144
ROLLING WINDOW OPERATIONS	101	GROUPING AND AGGREGATION	145
BINNING OPERATIONS	102	MERGING AND JOINING DATAFRAMES	147
ARRAY MASKS AND MASKED ARRAYS	102	RESHAPING DATAFRAMES	148
ARRAY INPUT/OUTPUT OPERATIONS	103	APPLYING FUNCTIONS TO DATAFRAMES	149
DATE AND TIME OPERATIONS	104	STRING OPERATIONS ON DATAFRAMES	151
ERROR HANDLING IN NUMPY OPERATIONS	105	TIME SERIES OPERATIONS	152
MATHEMATICAL CONCEPTS	105	CATEGORICAL DATA OPERATIONS	153
REVIEW OF BASIC ARITHMETIC OPERATIONS	105	ADVANCED INDEXING TECHNIQUES	154
MORE UNIVERSAL FUNCTIONS	106	COMBINING DATAFRAMES	155
BASIC MATHEMATICAL FUNCTIONS	106	HANDLING DUPLICATES	157
TRIGONOMETRIC FUNCTIONS	107	DATA TYPE CONVERSIONS	158
STATISTICAL FUNCTIONS	108	COMPUTATION AND DESCRIPTIVE STATISTICS	160
LINEAR ALGEBRA OPERATIONS	108	WORKING WITH LARGE DATAFRAMES	161
FOURIER TRANSFORMS	109	ADVANCED DATAFRAME METHODS	162
RANDOM NUMBER GENERATION AND PROBABILITY	110	BEST PRACTICES AND COMMON PITFALLS	163
POLYNOMIAL OPERATIONS	111	IMPORTING AND EXPORTING DATA	165
NUMERICAL INTEGRATION	112	CSV (COMMA-SEPARATED VALUES) FILES	166
NUMERICAL DIFFERENTIATION	112	EXCEL FILES	167
COMPLEX NUMBER OPERATIONS	113	JSON (JAVASCRIPT OBJECT NOTATION) FILES	168
REDUCTION OPERATIONS	114	HTML FILES	170
SLIDING WINDOW OPERATIONS	114	WEB APIs AND JSON DATA	171
INTERPOLATION	115	CHUNKING LARGE FILES	172
INTEGRATION WITH OTHER SCIENTIFIC PYTHON LIBRARIES	116	HANDLING ENCODINGS	173

PARSING DATES AND TIMES	174	HANDLING MISSING DATA IN GROUPED OPERATIONS	232
HANDLING MISSING DATA DURING IMPORT	175	ADVANCED GROUPING PATTERNS	234
HANDLING INCONSISTENT DATA	177	COMBINING GROUPING WITH OTHER PANDAS OPERATIONS	235
EXPORTING SUBSETS OF DATA	178	COMMON PITFALLS AND HOW TO AVOID THEM	236
BEST PRACTICES FOR DATA I/O	179	BEST PRACTICES FOR EFFICIENT AND READABLE GROUPBY OPERATIONS	237
IMPORTING DATA FROM OTHER PYTHON OBJECTS	180		
EXPORTING TO OTHER PYTHON OBJECTS	181		
BASIC DATA MANIPULATION	183	MERGING AND CONCATENATING DATA	239
INTRODUCTION TO DATA MANIPULATION	183	INTRODUCTION TO DATA COMBINATION	239
ACCESSING AND VIEWING DATA	184	CONCATENATION BASICS	240
SELECTING COLUMNS	185	ADVANCED CONCATENATION TECHNIQUES	241
FILTERING ROWS	186	MERGING FUNDAMENTALS	242
ADDING AND REMOVING COLUMNS	187	ONE-TO-ONE, ONE-TO-MANY, AND MANY-TO-MANY RELATIONSHIPS	244
RENAMING COLUMNS	189	MERGING ON INDEX	245
HANDLING MISSING DATA	190	HANDLING DUPLICATE COLUMNS	246
CHANGING DATA TYPES	191	INDICATOR COLUMN	247
STRING MANIPULATION	193	MERGING MULTIPLE DATAFRAMES	248
CATEGORICAL DATA	194	COMBINING MERGE AND CONCAT OPERATIONS	249
SORTING DATA	195	JOIN OPERATIONS	250
GROUPING AND AGGREGATION	196	HANDLING MISSING DATA IN MERGES AND CONCATENATIONS	252
MERGING AND JOINING DATA	197	CROSS JOINS (CARTESIAN PRODUCT)	253
RESHAPING DATA	199	TIME SERIES SPECIFIC OPERATIONS	254
HANDLING DUPLICATES	200	ADVANCED MERGE TECHNIQUES	255
APPLYING FUNCTIONS	202	COMMON PITFALLS AND TROUBLESHOOTING	256
BINNING DATA	203	BEST PRACTICES FOR MERGING AND CONCATENATING	257
WORKING WITH TIME SERIES DATA	204		
BASIC CALCULATIONS	205	MATPLOTLIB	259
HANDLING OUTLIERS	206	OVERVIEW OF MATPLOTLIB	259
DATA NORMALIZATION AND SCALING	208	MATPLOTLIB ARCHITECTURE	259
CREATING DERIVED VARIABLES	209	BASIC PLOTTING WITH PYPLOT	261
RENAMING AND REORDERING COLUMNS	210	FIGURE AND AXES OBJECTS	262
WORKING WITH MULTI-LEVEL INDEXES	211	CUSTOMIZING PLOT APPEARANCE	263
STRING PARSING AND EXTRACTION	213	SAVING PLOTS	264
TRANSPOSING DATAFRAMES	214	PLOT TYPES OVERVIEW	265
COMBINING DATAFRAMES	215	WORKING WITH TEXT IN PLOTS	267
HANDLING DATA CONSISTENCY	216	COLOUR MAPS AND COLOURBARS	268
COMMON PITFALLS AND BEST PRACTICES	217	LEGEND CREATION AND CUSTOMIZATION	269
GROUPING DATA	220	OBJECT-ORIENTED INTERFACE	270
INTRODUCTION TO DATA GROUPING	220	ANIMATION BASICS	272
THE GROUPBY OBJECT IN PANDAS	221	INTEGRATING WITH PANDAS	273
BASIC GROUPING OPERATIONS	222	CUSTOMIZING STYLES	274
ADVANCED GROUPING TECHNIQUES	223	3D PLOTTING INTRODUCTION	276
HIERARCHICAL INDEXING AND MULTI-LEVEL GROUPING ..	225	INTERACTIVE PLOTS	277
TRANSFORMATION AND FILTERING OPERATIONS	226	BEST PRACTICES	278
GROUP ITERATION AND ANALYSIS	227	TROUBLESHOOTING COMMON ISSUES	279
GROUPED OPERATIONS ON SPECIFIC COLUMNS	228		
RESHAPING DATA USING GROUPING	229	CREATING DIFFERENT TYPES OF PLOTS	282
GROUPING WITH CATEGORICAL DATA	230	LINE PLOTS	282
GROUP-WISE OPERATIONS	231		

SCATTER PLOTS	283	WHAT IS GEOPANDAS?	342
BAR PLOTS	284	INSTALLATION AND SETUP	343
HISTOGRAMS	286	CORE DATA STRUCTURES IN GEOPANDAS	343
BOX PLOTS	287	SUPPORTED GEOMETRY TYPES	344
PIE CHARTS	288	CREATING GEOMETRIC OBJECTS	345
AREA PLOTS	289	READING AND WRITING SPATIAL DATA	346
CONTOUR PLOTS	290	COORDINATE REFERENCE SYSTEMS (CRS)	347
HEATMAPS	291	BASIC ATTRIBUTE OPERATIONS	348
POLAR PLOTS	293	BASIC SPATIAL OPERATIONS	349
3D SURFACE PLOTS	294	SPATIAL JOINS	351
FILLED PLOTS	295	DISSOLVING AND AGGREGATING DATA	352
ERRORBAR PLOTS	296	EXPLORING THE SPATIAL EXTENT OF DATA	353
RADAR CHARTS (SPIDER PLOTS)	297	WORKING WITH PROJECTIONS	354
VIOLIN PLOTS	299	GEOPANDAS AND OTHER LIBRARIES	355
GANTT CHARTS	300	PERFORMANCE CONSIDERATIONS	356
WAFFLE CHARTS	301	REAL-WORLD EXAMPLES	357
SANKEY DIAGRAMS	302	WORKING WITH GEOMETRIC DATA	359
COMBINATION PLOTS	303	REVIEW OF BASIC GEOMETRY TYPES	359
GEOGRAPHIC PLOTS	304	ADVANCED GEOMETRY CREATION	360
ANIMATION OF DIFFERENT PLOT TYPES	305	ACCESSING AND MANIPULATING GEOMETRY PROPERTIES	361
CUSTOMIZATION TECHNIQUES	307	GEOMETRIC OPERATIONS	362
SAVING AND EXPORTING	309	SPATIAL RELATIONSHIPS AND PREDICATES	364
BEST PRACTICES	310	MEASURING GEOMETRIC PROPERTIES	365
INTRODUCTION TO SEABORN	313	TOPOLOGY OPERATIONS	366
OVERVIEW OF SEABORN	313	WORKING WITH LINES AND NETWORKS	368
SETTING UP SEABORN	314	POLYGON OPERATIONS	369
SEABORN PLOTTING FUNCTIONS OVERVIEW	314	MULTI-PART GEOMETRY HANDLING	370
WORKING WITH BUILT-IN DATASETS	315	COORDINATE TRANSFORMATIONS WITHIN GEOMETRIES	371
RELATIONAL PLOTS	316	GEOMETRY SIMPLIFICATION AND GENERALIZATION	372
CATEGORICAL PLOTS	317	RASTER-VECTOR OPERATIONS	374
DISTRIBUTION PLOTS	319	3D GEOMETRIES	376
REGRESSION PLOTS	320	GEOMETRY VALIDATION AND REPAIR	377
MATRIX PLOTS	321	ADVANCED SPATIAL JOINS AND OVERLAYS	378
PAIR PLOTS AND JOINT PLOTS	322	CUSTOM GEOMETRIC OPERATIONS	379
FACET GRIDS AND SUBPLOT GRIDS	323	WORKING WITH LARGE GEOMETRIC DATASETS	380
CUSTOMIZING SEABORN PLOTS	324	GEOMETRY SERIALIZATION AND DESERIALIZATION	381
WORKING WITH PANDAS DATAFRAMES	325	INTERFACING WITH OTHER GEOSPATIAL LIBRARIES	383
STATISTICAL ESTIMATION AND ERROR BARS	326	REAL-WORLD APPLICATIONS	384
COLOUR PALETTE CUSTOMIZATION	327	VISUALIZATION WITH GEOPANDAS	386
ADVANCED CUSTOMIZATION	329	INTRODUCTION TO GEOPANDAS VISUALIZATION	386
COMBINING SEABORN WITH MATPLOTLIB	330	BASIC PLOTTING WITH GEOPANDAS	387
TIME SERIES VISUALIZATION	331	CUSTOMIZING PLOT AESTHETICS	388
CATEGORICAL DATA ENCODING	332	PLOTTING MULTIPLE LAYERS	389
MULTI-PLOT LAYOUTS	333	CHOROPLETH MAPS	391
SAVING AND EXPORTING SEABORN PLOTS	334	CATEGORICAL PLOTTING	392
SEABORN THEMES	336	POINT PATTERN MAPS	393
STATISTICAL ANNOTATIONS	337	HEAT MAPS AND DENSITY PLOTS	394
BEST PRACTICES	338	LINE AND NETWORK VISUALIZATION	395
SEABORN VS. MATPLOTLIB: WHEN TO USE EACH	339	ADDING MAP ELEMENTS	396
INTRODUCTION TO GEOPANDAS	342		

CUSTOMIZING PROJECTIONS IN PLOTS	398
INTERACTIVE MAPPING WITH GEOPANDAS	399
ANIMATIONS AND TIME SERIES	400
3D VISUALIZATION.....	401
CUSTOMIZING MAP LAYOUTS	403
THEMATIC MAPPING TECHNIQUES.....	404
ADVANCED STYLING TECHNIQUES.....	405
VISUALIZING UNCERTAINTY AND DATA QUALITY	407
EXPORTING AND SHARING MAPS.....	408
INTEGRATING WITH OTHER VISUALIZATION LIBRARIES	410
REAL-WORLD VISUALIZATION CASE STUDIES	411
COMMON PITFALLS AND HOW TO AVOID THEM.....	413
ADVANCED GEOPANDAS CONCEPTS.....	415
ADVANCED DATA STRUCTURES IN GEOPANDAS	415
SPATIAL INDEXING AND QUERY OPTIMIZATION	416
ADVANCED COORDINATE REFERENCE SYSTEMS (CRS).....	418
TOPOLOGY OPERATIONS AND ANALYSIS	419
ADVANCED SPATIAL JOINS AND RELATIONSHIPS	420
RASTER-VECTOR INTEGRATION	422
TIME SERIES ANALYSIS WITH GEOSPATIAL DATA	423
NETWORK ANALYSIS.....	425
SPATIAL STATISTICS AND GEOSTATISTICS	426
CUSTOM GEOMETRY OPERATIONS	428
3D AND VOLUMETRIC ANALYSIS.....	430
TEMPORAL AND MOVING OBJECT ANALYSIS	432
SPATIAL OPTIMIZATION PROBLEMS	433
ADVANCED OVERLAY AND SPATIAL OPERATIONS.....	434
GEOSPATIAL DATA STREAMING AND REAL-TIME ANALYSIS	436
WORKING WITH NON-STANDARD GEOMETRIC DATA.....	437
ADVANCED PROJECTION HANDLING	439
HANDLING UNCERTAINTY IN SPATIAL ANALYSIS	440



Introduction

Hello and welcome to 'Mastering Python: From Basics to Machine Learning', a comprehensive course designed to teach you everything you'll want to know about Python and more. Whether you're a complete beginner or looking to enhance your Python skills, there's something here for everyone.

We will start from the ground up, taking our time to fully understand basic and intermediate topics, building a solid foundation before moving onto more advanced and complex concepts. At the end of the course, you will be confident in writing advanced applications in Python all by yourself.

The course is split into several key sections, broken down into a series of lectures and practical exercises:

- Python Basics such as data structures and control flows
- Intermediate topics such as object-oriented programming and error handling
- Advanced data analytics and visualization
- Key libraries, such as NumPy and Pandas and also GeoPandas and Folium
- Web Development and web scraping
- Machine Learning, where we'll look at supervised, unsupervised and deep learning using PyTorch and scikit-learn

In each section, you'll get lots of hands-on exercises and we'll build practical projects together to solidify your understanding.

Now, why exactly should you choose to learn Python? Simply put, Python is easy to learn, it's versatile, and

very powerful. It's known for its simple syntax, which closely resembles the English language, making it a great choice for beginners and experts alike.

Python is used in many different fields, from web development - sites like Instagram, Pinterest, or Dropbox - and data science to artificial intelligence - Siri and Alexa both run Python behind the scenes - and automation. It comes with a large standard library and enjoys the support of a huge community. It consistently ranks as one of the most popular programming languages in surveys and indexes. In fact, it often tops the charts in surveys of developers and data scientists.

By the time you finished this course, you will:

- Have a good understanding of Python syntax and general programming concepts
- Master data structures and control structures
- Get to grips with file handling, error handling, and object-oriented programming
- Analyse and visualize data
- Build a web application using Flask
- And develop a solid foundation for advanced data science and machine learning concepts

But this course is more than just ticking off topics. It's just as much about developing a programmer's and data scientist's mindset. You'll learn to ask the right questions and to approach problems systematically.

To get the most out of this course, I strongly recommend that you

- Set aside regular time for learning - 30 to 60 minutes a day will do.
- Code along with me. There is no better way to learn than to do. Make sure that you download all of the extra materials from the course's website for this.
- We also have a great Discord community, make sure to engage with fellow learners
- And finally, try to use your new skills on real-world problems. Personal projects are great for cementing knowledge and they make great additions to your portfolio

And there we have it. You will learn everything from Python basics to advanced topics in great detail and lots of things in between, get to do plenty of hands-on exercises and projects all while gaining lots and lots practical experience. Happy coding!

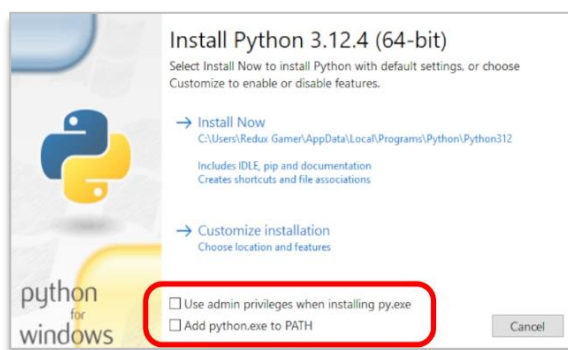


Installation and Setup

First things first, let's download Python. Head on over to the Python website at python.org and click on the Downloads tab. Here, you'll find the latest version of Python. The website will automatically suggest the best version for your operating system. Click the download button and wait for the download to complete.

Once the download has completed, locate the downloaded installer, double-click it, and follow the instructions along.

On Windows it is very important that you select the two checkboxes "Use admin privileges" and "Add python.exe to PATH" at the start of the installation process.



Click on 'Install Now' and let the installer do its thing. Depending on your machine this might take a couple

of minutes. Once it's done, you'll see a success message and you've just installed Python! It's as simple as that.

Great, now with Python installed, let's make sure it's working properly. Open a terminal on Mac or command prompt on Windows and type `python --version` or `python3 --version`. If Python is installed correctly, you'll see the version number displayed. Yay!

Let's take this a step further. Type `python` into your terminal and hit enter. You should see the Python interactive shell, which looks something like this:

```
(base) X:\>python
Python 3.11.5 | packaged by Anaconda, Inc. | (main, Sep 11 2023, 13:26:23)
[MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
```

These three chevrons `>>>` are Python's way of saying 'I'm listening'. Let's give it something to do. Type `print('Hello, Python!')`.

If you see `Hello, Python!` printed back to you, congratulations! You've just run your first Python command and you're all set to go!

But before we get into the nitty gritty of Python syntax we'll set up our development environment to make coding even more enjoyable. Think of this as preparing your workspace.

Installing VSCode

In this course we'll use one of the most popular IDEs – Visual Studio Code, or VSCode for short. It's free, powerful, and highly customizable.

Head over to code.visualstudio.com and download the installer for your operating system. Once downloaded, run the installer and follow the on-screen instructions. It's as simple as clicking 'Next' a few times and hitting 'Install.'

Once installed, open VSCode to make sure that it's running fine.

Setting up VSCode

With VSCode running, let's set it up for Python development. Go to the Extensions view by clicking the square icon in the sidebar. Search for 'Python' and install the extension provided by Microsoft. This

extension adds rich support for Python, including features like IntelliSense, linting, and debugging.

Although we'll be using VSCode, there are, of course, other popular choices for programming as well such as JetBrains PyCharm (free for students), Sublime, Atom, or Spyder. Feel free to try a few and see which one fits best!

Now it's your turn to write your first script. After setting up VSCode, let's open a folder where you'll keep your Python scripts for this course.

Create a new Python file called, `hello.py` inside this folder and type in the classic first program: `print('Hello, World!')`. Python files have the extension `.py` as you can see. Save the file and run it to see your code in action.

To run your code, let's open the terminal inside of VSCode. Go to View → Terminal, which will open the terminal. In the terminal you can run your code by typing: `python hello.py`

Hopefully you'll see "Hello, World!" printed in the terminal.

🔥 If you are on Mac or Linux and this didn't work, type `python3 hello.py` to tell your system that you're using Python 3.

Once you've got this running, congratulations! You've just set up your Python development environment and run your first proper Python program. If you want to customize the way that VSCode looks, there are plenty of fantastic tutorials out there to help.



Python Syntax

Python is designed with readability in mind, which makes it unique among programming languages. The language emphasizes writing clean and readable code, allowing developers to express their ideas clearly and concisely. By following simple guidelines, you can write code that is easier to understand, maintain, and share with others, leading to more effective collaboration and development.

Okay, it's time to dive into the very heart of Python: its syntax. In this section, we'll cover the basic syntax and structure of Python, show you how to write and run your first Python program, introduce comments in Python, and walk through some examples and exercises, teaching you the basic building blocks of Python code. But, before we jump in, however, just a quick heads up. The first few sections of this course might feel a little dry and very structured. That's perfectly normal. It's very important that we get the basics right, to fully understand the intermediate and advanced topics. Please, don't let that dissuade you! As soon as we've gotten a strong foundation, like how to define variables and basic syntax out of the way, we'll get into the exciting dynamic stuff. Let's get right to it and fire up your VSCode as we did in the last chapter.

A tradition in programming is to start with a simple "Hello, World!" program, as we have seen already. It is a great way to get familiar with the basic syntax of the language and to make sure that everything is set up properly. To write this program in Python, we used the `print()` function, which is a built-in function that outputs text to the console. The `print` command tells

Python to display the text "Hello, World!" on the screen. It's a simple yet powerful way to begin your coding journey. Do you remember how to do it? Try creating a new file and writing the hello world program and then running it. Go back to the previous chapter if you need to.

print() options

One of the functions that we have already seen, is the print function. The print() function is one of the most commonly used functions in Python, and it's essential for displaying information to the user. When you call print(), you can pass one or more arguments, which are the values you want to print. These values can be strings, numbers, or other data types. You can also print multiple items by separating them with commas, like `print("Hello,", "World!")`, which will print each item separated by a space. Understanding how to use the print() function is crucial as it will be your primary tool for debugging and communicating with users through your programs. As you go through the course we'll come across it time and time again – it will become your best friend.

The “end” parameter in print()

The print() function in Python has several optional parameters that allow you to control its behaviour. One such parameter is `end`, which defines what is printed at the end of the output. By default, `print()` ends with a newline character, which means each print() statement starts on a new line. You can change this behaviour by setting the end parameter to a different string. For example:

```
print("Usain", end="-")
print("Bolt")
```

This code will output `Usain-Bolt` on the same line because the end parameter was set to an exclamation mark instead of the default newline.

The “sep” parameter in print()

Another useful parameter in the `print()` function is `sep`, which stands for separator. This parameter controls the string inserted between multiple arguments passed to `print()`. By default, `sep` is a space. You can change this to any string you prefer.

This feature is particularly useful when you need to format the output in a specific way, such as generating comma-separated values (CSV).

The help() Function

Python comes with a built-in `help()` function that provides useful information about Python modules, functions, classes, and more. To use it, simply call `help()` with the name of the item you need help with as an argument. For example, `help(print)` will display documentation about the `print()` function, including its parameters and usage examples. This function is incredibly helpful when you need to understand the functionality of Python's built-in features or when exploring new libraries. Make sure to try out using `help()`.

Comments in Python

Comments are an integral part of writing clear and understandable code. This is not only for you to tell others what you have done, but also for yourself in the future. So, comments are essential for documenting your code, making it easier for you (and others) to understand the logic and purpose behind it when revisiting the code later. There are two ways to write comments in python.

Single-line comments (#)

Single-line comments are created using the `#` symbol. Everything after the `#` on that line is ignored by the Python interpreter. This allows you to add explanations or notes about what the code does. For example, `# This is a single-line comment` is a comment that won't affect the code execution.

```
# Saying hello to the world
print("Hello, World!") # Another comment
```

Multi-line comments (""" """)

For more extensive explanations or documentation, you can use multi-line comments, which are created using triple quotes, either `"""` or `'''`. These are particularly useful for adding detailed descriptions or explanations at the beginning of a script or function. Multi-line comments help in providing a thorough context for your code, which is beneficial for larger projects or when collaborating with others.

```
"""
The hello world program to test the system setup

Nothing special to see here.
"""
```

Commenting is a best practice that significantly enhances the readability and maintainability of your

code. Good comments should be clear, concise, and relevant. They should explain why certain decisions were made or describe complex logic. However, avoid over-commenting, which can clutter your code and aim for a balance where comments add value and clarity without becoming redundant instead. Consistent and thoughtful commenting will go a long way.

Line Continuation

In Python, if you have a long line of code that exceeds the recommended line length you can break it into multiple lines using the backslash `\` for explicit line continuation. This helps improve the readability of your code. For example:

This method ensures that Python understands the code is continued on the next line, making your code cleaner and easier to read.

```
total = 1 + 2 + 3 + \
        4 + 5 + 6
```

Python also allows for implicit line continuation within parentheses `()`, brackets `[]`, and braces `{}`. This means you can write code across multiple lines without needing a backslash. For instance:

```
total = (1 + 2 + 3 +
        4 + 5 + 6)
```

This approach is often more readable and less error-prone than using explicit line continuation. It's particularly useful for function calls, array definitions, and dictionary entries, where breaking lines makes the structure more apparent. We will look at all of those in due time.

Case sensitivity

And finally, Python is a case sensitive language. This means that functions, such as `print()`, need to be written in the case that they are defined. Try writing the hello world programming and making `print()` upper case.

You see, that didn't work, did it? The same is true for variables that we use. We'll get to variables in the next chapter.

That's it for the basic syntax of Python. Remember that Python is designed to be readable and proper indentation is essential. It's case sensitive and you should get into the habit of fully documenting your

code as you go along. Finally, the print function will be your best friend!



Variables and Data Types

Variables are fundamental building blocks in programming. They act as storage containers for data that can be manipulated and retrieved as needed. When you create a variable, you are essentially reserving a memory location to store a value, which can be of various data types such as integers, strings, or floats. This concept allows you to write more flexible and dynamic code by referring to these stored values through descriptive names rather than using the data directly.

Think of variables as labelled boxes that can hold different types of data. For example, you might have a box labelled "age" that contains the number 25, or a box labelled "name" that contains the string "Alice." By labelling these boxes, you can easily access and modify their contents without worrying about the underlying data structure. This abstraction is crucial for managing complex programs and makes your code more readable and maintainable.

Variables are essential because they allow you to store, manipulate, and retrieve data efficiently. Without variables, you would need to hard-code values directly into your programs, making them inflexible and difficult to modify. Variables enable you to write more dynamic and reusable code, as you can change the value of a variable without altering the

program's logic. They also help in organizing your code, making it easier to understand and debug.

Creating Variables

Creating a variable is super straightforward. You simply choose a name for your variable and use the assignment operator `=` to assign a value to it. For example, `age = 25` creates a variable named `age` and assigns it the value 25. This process is known as variable assignment, and it allows you to store different types of data in a way that can be easily referenced and manipulated later in your code.

The assignment operator (=)

The assignment operator `=` is used to assign values to variables. It's important to understand that this operator does not mean equality, as it does in mathematics. Instead, it takes the value on the right side and stores it in the variable on the left side. For instance, in the statement `name = "Alice"`, the string "Alice" is assigned to the variable `name`. This distinction is crucial to avoid confusion, especially when dealing with comparisons and logical expressions.

Variable Naming Rules and Conventions

In Python, variable names must start with a letter (a-z, A-Z) or an underscore (`_`), followed by letters, numbers, or underscores. Names like `1st_name` are invalid because they start with a number, while names like `first_name` and `_firstname` are perfectly valid. Reserved keywords such as `class` cannot be used as variable names because they have special meaning in Python. More on this in due course.

PEP 8 naming conventions

According to PEP 8, Python's style guide, variable names should be in lowercase with words separated by underscores to improve readability. This naming style is known as snake case. For example, `first_name` is preferred over `FirstName` or `firstName`. Consistently following these conventions makes your code easier to read and maintain.

Reserved keywords that can't be used as variable names

Python has a set of reserved keywords that have special meanings and cannot be used as variable names. These include words like `if`, `else`, `for`, `while`, and `class`. Using these as variable names will result

in a syntax error. It's essential to avoid these keywords when naming your variables.

Naming Best Practices

Using descriptive variable names is a best practice that significantly enhances the readability of your code. Instead of naming a variable `x`, name it `user_age` if it stores the age of a user. Descriptive names make it clear what the variable represents, reducing confusion and making your code more maintainable.

Avoiding single-letter names (except in specific cases)

While single-letter names like `x` or `i` can be useful in small, simple contexts like loop counters, they should generally be avoided in favour of more descriptive names. This practice makes your code easier to understand. However, in cases where the purpose is clear and the scope is limited, such as `i` in a for loop, single-letter names can be appropriate.

Basic Data Types for Variables

There are four basic variable types in Python: integers, floating-point numbers, booleans, and strings.

Integers (int)

Integers are whole numbers without a fractional component. They can be positive, negative, or zero. In Python, you can create an integer by assigning a whole number to a variable, like `age = 25`.

Floating-point numbers (float)

Floats are numbers with a decimal point, used to represent real numbers that require precision. For example, `shoe_size = 42.5` creates a float variable in Python. Floats are essential for calculations involving fractions and precise measurements.

Strings (str)

Strings are sequences of characters enclosed in quotes. They can be single quotes ('hello') or double quotes ("hello"). Strings are used to store text, and Python provides various methods to manipulate them, such as concatenation and slicing. We will see all of those in the next chapter.

Booleans (bool)

Booleans represent one of two values: True or False. They are commonly used in conditional statements and logical operations. For example, `is_student = True` assigns a boolean value to the variable `is_student`.

Variable Types

The `type()` function is used to determine the data type of a variable. It returns the type of the object passed to it. For instance, if you have a variable `age = 25`, calling `type(age)` will return `<class 'int'>`. This function is particularly useful for debugging and understanding what type of data a variable holds.

```
print(type(age))
print(type(height))
print(type(name))
print(type(is_student))
```

The `isinstance()` function is used to check if an object is an instance of a specific. It returns True if the object is an instance or subclass of the class, and False otherwise.

Variable Reassignment

In Python, you can change the value of a variable by simply assigning a new value to it. For example, if you initially have `x = 10`, you can later reassign it to a different value, such as `x = 20`. This flexibility allows you to update variables as needed throughout your program.

Dynamic typing

Dynamic typing means that the type of a variable is determined at runtime and can change during the execution of the program. Unlike statically-typed languages where you must declare the type of a variable, Python allows you to assign any type of value to a variable without specifying its type. This flexibility simplifies coding but requires careful handling to avoid runtime errors.

```
x = 10
x = "Hello"
```

In statically-typed languages like C or Java, you must declare the type of a variable before using it, and the type cannot change. For example, in Java, you would declare an integer variable as `int age = 25`; Python's dynamic typing, on the other hand, allows you to write `age = 25` without declaring the type, and you

can later reassign `age` to a different type, such as `age = "twenty-five"`.

Multiple Assignment

Python allows you to assign values to multiple variables in a single line. This feature, known as multiple assignment, can make your code more concise and readable. For example, `a, b, c = 1, 2, 3` assigns the values 1, 2, and 3 to `a`, `b`, and `c`, respectively.

You can also use multiple assignment to swap the values of two variables without needing a temporary variable. For example, `a, b = b, a` swaps the values of `a` and `b`. This technique is both elegant and efficient, making it a common practice in Python programming.

```
a, b, c = 1, 2, 3
a, b = b, a
```

Constants

Constants are variables whose values should not change once assigned. They are used to represent fixed values that do not change throughout the program. While Python does not have built-in constant types, it is a convention to use all uppercase letters to signify that a variable should be treated as a constant.

According to PEP 8, the naming convention for constants is to use all uppercase letters with underscores separating words. For example, `PI = 3.14159` and `GRAVITY = 9.8` are constants representing fixed values. While Python does not enforce immutability, using this convention helps signal to other programmers that these values should not be changed.

```
PI = 3.14159
GRAVITY = 9.8
```

print() function with variables

The `print()` function can be used to display the values of variables. For example, `print(name)` will output the value of the variable `name`. This is a straightforward way to show the current state of your variables and is especially useful for debugging.

You can combine text and variables in the `print()` function using concatenation or formatted strings. This is something that we will look at in the next chapter.

The None value

In Python, `None` is a special constant that represents the absence of a value or a null value. It is an object of its own data type, `NoneType`. You can assign `None` to a variable to indicate that it does not currently hold a meaningful value. For example, `result = None` initializes `result` with no value. This is useful for initializing variables or indicating a special condition, such as the end of a list or an uninitialized state.

Type Conversion

Type conversion functions like `int()`, `float()`, and `str()` allow you to convert variables from one type to another. For example, you can convert a string to an integer using `int("123")` or a float to a string using `str(3.14)`. These conversions are essential for handling user input, performing calculations, and managing different data types.

Type conversion is necessary when you need to perform operations that require specific data types. For instance, if you receive numeric input as a string from the user, you must convert it to an integer or float before performing arithmetic operations. Type conversion ensures that your variables are in the correct format for the operations you need to perform, preventing errors and improving code robustness.

```
# Get age from user
age = input("Age: ")
age_int = int(age)

# Increase age by 1
age_int = age_int + 1
```

Since `input()` returns a string, you often need to typecast the input to the desired data type. Here, the user's input is converted from a string to an integer using the `int()` function. This typecasting is essential for performing numeric operations on the input value.

Checking Variable Existence

To check if a variable exists in the global scope, you can use the `in` keyword with the `globals()` function. This code checks if `age` and `name` are defined in the global scope. Using this method helps you determine whether a variable has been declared and can prevent errors in your programs.

```
print('age' in globals())
```

Variables in Simple Expressions

Variables can be used in boolean expressions to evaluate conditions. For example:

```
age = 20
is_adult = age >= 18
print(is_adult) # Output: True
```

In this code, the variable `is_adult` is assigned the result of the boolean expression `age >= 18`, which evaluates to `True`. Using variables in boolean expressions is common in conditional statements and loops.

Python provides several comparison operators for evaluating variables, such as `==`, `!=`, `>`, `<`, `>=`, and `<=`. For example:

```
a = 5
b = 10
print(a < b) # Output: True
print(a == b) # Output: False
```

These operators allow you to compare variables and perform logical operations based on their values.

Underscore Variable

The underscore `_` is commonly used as a throwaway variable in Python, indicating that a value is being intentionally ignored. For example, in a loop:

```
# print "Hello" five times
for _ in range(5):
    print("Hello")
```

In this code, `_` is used to ignore the loop variable because it is not needed in the loop body. This practice improves code readability by signalling that the variable's value is not important.



Strings

In Python, a string is simply a sequence of characters. It can contain letters, numbers, symbols, or even spaces.

```
favourite_food = "Pepperoni pizza"
```

An important characteristic of strings is that they're immutable. This means once a string is created, you can't change its contents.

To make strings, you can use characters from virtually any language or symbol system. For example:

```
unicode_string = "こんにちは, world! 🌐"
print(unicode_string)
```

Strings are incredibly important in programming. Any time you work with text, you're working with strings. When you display information to users or receive input from them, you're often dealing with strings.

```
name = "Alice"
greeting = "Hello, " + name + "!"
print(greeting)
```

Strings are also often used to store and transfer data, especially when working with files or sending data over networks. When you do this, strings can be used to represent and work with other types of data.

```
age = 25
age_string = str(age)
print("I am " + age_string + " years old.")
```

Creating Strings

Now that we understand what strings are, let's explore the different ways to create them. You can create strings using either single quotes or double quotes. They're interchangeable:

```
single_quoted = 'Hello, World!'
double_quoted = "Hello, World!"
print(single_quoted)
print(double_quoted)
```

Both of these will output the same thing. So why have two options? Well, it gives us flexibility when our string itself contains quotes:

```
sentence = "He said, 'Python is amazing!'"
another_sentence = 'She replied, "I totally agree!"'
print(sentence)
print(another_sentence)
```

As for coding style, the most important thing is to be consistent. Pick one style and stick with it throughout your project.

When you need a string that spans multiple lines, you can use triple quotes. You can use either three single quotes or three double quotes:

```
multi_line = '''This is a string
that spans across
multiple lines.'''
```

Triple quotes preserve formatting and line breaks. They're also commonly used to document more complex code.

String Indexing

Access individual letters in a string is called string indexing and allows us to access individual characters in a string. Since strings are sequences of characters, we can access each character using its index by writing `string[index]`. Indexing in Python is zero-based, which means the first character is at index 0:

```
text = "Python"
first_char = text[0]
second_char = text[1]
print(first_char) # Outputs: P
print(second_char) # Outputs: y
```

Python allows both positive and negative indexing. Positive indices start from 0 and go up to `length-1`. Negative indices start from `-1` (last character) and go down to `-length`:

```
text = "Python"
print(text[0]) # First character (P)
print(text[5]) # Last character (n)
print(text[-1]) # Last character (n)
print(text[-6]) # First character (P)
```

As you can see, `text[5]` and `text[-1]` both give us the last character, while `text[0]` and `text[-6]` both give us the first character.

If you try to access an index that doesn't exist, Python will raise an `IndexError`. We'll look at Errors in much more detail in a later lecture. This happens when the index is too large (greater than or equal to the length of the string) or too small (less than negative length):

```
text = "Python"
# print(text[6]) # This would raise an IndexError
# print(text[-7]) # This would also raise an IndexError
```

To avoid these errors, always make sure your index is within the valid range. One way to do this is to use the `len()` function to check the length of the string:

```
text = "Python"
length = len(text)
if length >= 6:
    print(text[5])
```

We haven't covered conditional statements yet, but this gives you an idea of how you might handle potential index errors in the future.

Remember, mastering string indexing is crucial for many string operations we'll learn about in the future.

String Slicing

Let's take string indexing one step further and let's explore string slicing, which allows us to extract portions of a string.

The basic syntax for slicing is `[start:end]`.

- The start index is inclusive (included in slice)
- The end index is exclusive (not included)
- If you omit start or end, Python uses default values of beginning and end of the string.

Let's see this in action:

```
text = "Python Programming"
slice1 = text[0:6] # Characters from index 0 to 5
print(slice1) # Outputs: Python

slice2 = text[7:18] # Characters from index 7 to 17
print(slice2) # Outputs: Programming
```

You can omit either the start or end index for more flexible slicing:

```
text = "Python Programming"

# Slicing from the beginning
print(text[:6]) # Equivalent to text[0:6]
# Outputs: Python

# Slicing to the end
print(text[7:]) # Equivalent to text[7:len(text)]
# Outputs: Programming

# Full string slice
print(text[:]) # Copies the entire string
```

You can also add a third parameter to the slice syntax: the step – `[start:stop:step]`. This defines the increment between each character in the slice.

```
text = "Python Programming"

# Positive step (default is 1)
print(text[0:18:2]) # Outputs: Pto rgamn

# Negative step
print(text[18:0:-1]) # Outputs: gnimmargorP nohty

# Skipping characters
print(text[::3]) # Outputs: Ph rrmn
```

A neat trick with slicing is that you can easily reverse a string using `[::-1]`:

```
text = "Python"
reversed_text = text[::-1]
print(reversed_text) # Outputs: nohtyP
```

This works because it starts from the end (default when start is omitted), goes to the beginning (default when end is omitted), and moves with a step of -1 (i.e., one character to the left each time).

String Concatenation

Now, let's look at string concatenation, which is the process of combining strings. We can simply use the `+` operator to concatenate strings:

```
first_name = "John"
last_name = "Doe"
full_name = first_name + " " + last_name
print(full_name) # Outputs: John Doe

greeting = "Hello, " + full_name + "!"
print(greeting) # Outputs: Hello, John Doe!
```

Note that when using `+`, both operands must be strings. If you try to concatenate a string with a non-string, Python will raise a `TypeError` – see those Errors again!

```
age = 30
# This would raise a TypeError:
# print("Age: " + age)

# Instead, convert the non-string to a string first:
print("Age: " + str(age))
```

Python also allows you to repeat a string using the `*` operator:

```
word = "Python"
repeated = word * 3
print(repeated) # Outputs: PythonPythonPython

line = "-" * 20
print(line) # Outputs: -----
```

You can combine repetition with concatenation:

```
border = "+" + "-" * 10 + "+"
print(border) # Outputs: +-----+
```

This is particularly useful for creating formatted output or simple text-based graphics. Remember, string concatenation and repetition are powerful tools for creating and manipulating text in Python.

String Methods

Python has many methods that allow us to manipulate and analyse strings easily. We'll cover the main ones here. There are plenty more and you can find information on them in the Python documentation.

`len()`: Getting the length of a string

The `len()` function returns the number of characters in a string:

```
text = "Python"
length = len(text)
print(length) # Outputs: 6

unicode_text = "Python 🐍"
print(len(unicode_text)) # Outputs: 8
                        # (7 letters + 1 emoji)
```

Note that `len()` counts Unicode characters, not bytes. This means emojis and other complex Unicode characters are counted as single characters.

`upper()`, `lower()`, `title()`, `capitalize()`

These methods change the case of strings:

```
text = "hello, world!"

print(text.upper()) # Outputs: HELLO, WORLD!
print(text.lower()) # Outputs: hello, world!
print(text.title()) # Outputs: Hello, World!
print(text.capitalize()) # Outputs: Hello, world!
```

They do not modify the original string; but return a new string. Non-alphabetic characters are left unchanged.

`strip()`, `lstrip()`, `rstrip()`

These methods remove whitespace (or specified characters) from strings:

```
text = " Python "
print(text.strip()) # Outputs: "Python"
print(text.lstrip()) # Outputs: "Python "
print(text.rstrip()) # Outputs: " Python"

text_with_stars = "***Python***"
print(text_with_stars.strip('*')) # Outputs: "Python"
```

They are especially useful for cleaning up user input or formatted data. Have a close look at the examples.

`replace()`

The `replace()` method substitutes occurrences of a substring with another string – like the `replace` function in Word.

```
text = "Hello, world!"
new_text = text.replace("world", "Python")
print(new_text) # Outputs: Hello, Python!

# Limiting replacements
text = "ha ha ha ha"
limited_replace = text.replace("ha", "HO", 2)
print(limited_replace) # Outputs: HO HO ha ha
```

Note that `replace()` is case-sensitive and returns a new string.

`startswith()`, `endswith()`

These methods check if a string starts or ends with a specific substring:

```
text = "Hello, world!"
print(text.startswith("Hello")) # Outputs: True
print(text.endswith("world!")) # Outputs: True

# Multiple options
print(text.startswith(("Hello", "Hi"))) # Outputs: True

# Optional start and end parameters
print(text.endswith("world", 0, 12)) # Outputs: True
```

`find()`, `index()`

These methods locate substrings within a string:

```
text = "Hello, world!"
print(text.find("world")) # Outputs: 7
print(text.index("world")) # Outputs: 7

print(text.find("Python")) # Outputs: -1
# print(text.index("Python")) # raises ValueError
```

However, there is an important difference between find and index when the sub-string does not exist. `find()` returns -1 if the substring is not found, while `index()` raises an exception.

`isalpha()`, `isdigit()`, `isalnum()`, `isspace()`

These methods check the content of strings. `isalpha()` returns True if all characters in the string are letters, while `isdigit()` returns True if all characters are numbers. You guessed it, `isalnum()` returns True if the string is made up of a combination of letters and numbers and `isspace()` returns True if the entire string is made up of spaces.

```
print("Hello".isalpha()) # Outputs: True
print("123".isdigit()) # Outputs: True
print("Hello123".isalnum()) # Outputs: True
print(" ".isspace()) # Outputs: True

print("").isalpha() # Outputs: False
# (empty string)
print("Hello!".isalpha()) # Outputs: False
# (contains non-letter)
```

These methods work with Unicode characters as well.

String Formatting with f-strings

Now, let's have a closer look at f-strings, or formatted strings, which are a powerful way to format strings in Python. F-strings provide a concise and convenient way to embed Python expressions inside strings. You simply put an 'f' or 'F' in front of the starting quote of the string and embed expressions inside string literals, using curly braces {}:

```
name = "Alice"
age = 30
print(f"My name is {name} and I'm {age} years old.")
```

In this example, `{name}` and `{age}` are replaced with the values of the name and age variables. F-strings make it easy to create formatted strings without complex string concatenation.

You can include any valid Python expression inside the curly braces:

```
x = 10
y = 20
print(f"The sum of {x} and {y} is {x + y}.")
```

You can even call functions within f-strings:

```
def double(n):
    return n * 2

number = 5
print(f"The double of {number} is {double(number)}.")
```

Don't worry about the syntax yet, we'll get to functions soon. F-strings also support formatting specifiers for more control over the output:

```
pi = 3.14159
print(f"Pi to 2 decimal places: {pi:.2f}")

for i in range(1, 4):
    print(f"Row {i:2d}: {i*i:3d}")

percentage = 0.8532
print(f"Percentage: {percentage:.2%}")
```

F-strings are the preferred string formatting tool and have several advantages over older formatting methods:

- ▣ They're more readable and concise.
- ▣ They're evaluated at runtime, so you can use variables and expressions directly.
- ▣ They're faster than other formatting methods.

Remember, f-strings are a powerful tool for creating formatted output in Python. They combine the readability of template strings with the power of Python expressions, making them an essential part of modern Python programming.

Escape Characters

Escape characters are special characters that are used to represent certain actions or characters that are difficult to type directly. Python supports several escape characters. Here are some of the most common ones:

```
# \n - Newline
print("Hello\nWorld") # Outputs: Hello
#                               World

# \t - Tab
print("Name:\tJohn") # Outputs: Name:      John

# \\ - Backslash
print("C:\\Users\\John") # Outputs: C:\Users\John

# \' - Single quote
print('It\'s a beautiful day') # Outputs: It's a beautiful day

# \" - Double quote
print("She said, \"Hello!\"") # Outputs: She said, "Hello!"
```

You can combine multiple escape characters in a single string:

```
print("Line 1\nLine 2\n\tIndented Line 3")
# Outputs:
# Line 1
# Line 2
#     Indented Line 3
```

```
file_path = "C:\\Program Files\\Python\\Python.exe"
# Outputs: C:\Program Files\Python\Python.exe
print(file_path)
```

Escape characters are particularly useful when you need to include quotes within strings or when you want to format your output with newlines and tabs.

String Immutability

I mentioned earlier that in Python, strings are immutable. This means that once a string is created, you cannot change its contents. Let's see what this means in practice:

```
text = "Hello"
# text[0] = "h" # This would raise a TypeError

# Instead, you create a new string
new_text = "h" + text[1:]
print(new_text) # Outputs: hello
```

Python implements strings this way because it allows for more efficient memory usage and caching.

When you perform operations that seem to modify a string, you're actually creating a new string:

```
original = "Hello"
modified = original.upper()

print(original) # Outputs: Hello
print(modified) # Outputs: HELLO
```

The 'in' Operator with Strings

The `in` operator allows you to check if a substring exists within a larger string. Here's how it works:

```
text = "Hello, World!"

print("Hello" in text) # Outputs: True
print("Python" in text) # Outputs: False
```

It is particularly useful for simple string searches:

```
email = "user@example.com"
print("@" in email) # Outputs: True

filename = "document.txt"
print(".txt" in filename) # Outputs: True
```

Remember, the `in` operator checks for the existence of a substring anywhere in the string. If you need to check for a substring at a specific position, you might need to use other string methods we've learned about earlier.

Splitting and Joining Strings

Splitting and joining strings is very useful when working with text data in Python and will most likely make use of this when working with data.

The `split()` method

The `split()` method allows us to break a string into parts based on a specified delimiter. Let's look at some examples:

```
# Default behavior (splitting on whitespace)
sentence = "Hello world how are you"
words = sentence.split()
print(words) # Outputs: ['Hello', 'world', 'how', 'are', 'you']

# Specifying a delimiter
csv_data = "apple,banana,cherry,date"
fruits = csv_data.split(",")
print(fruits) # Outputs: ['apple', 'banana', 'cherry', 'date']

# Controlling the number of splits
limited_split = csv_data.split(",", 2)
print(limited_split) # Outputs: ['apple', 'banana', 'cherry,date']
```

The `split()` method is very handy when you need to parse structured text data, like separating first and last name from a full name string.

The `join()` method

The `join()` method is the opposite of `split()` - it combines strings into a single string. Here's how it works:

```
# Basic joining
words = ['Hello', 'world', 'how', 'are', 'you']
sentence = " ".join(words)
print(sentence) # Outputs: Hello world how are you

# Using a different separator
fruits = ['apple', 'banana', 'cherry', 'date']
csv_data = ",".join(fruits)
print(csv_data) # Outputs: apple,banana,cherry,date

# Joining with a multi-character string
path_parts = ['C:', 'Users', 'YourName', 'Documents']
file_path = "\\ ".join(path_parts)
print(file_path) # Outputs: C:\Users\YourName\Documents
```

Note the syntax: we call `join()` on the string we want to use as the separator, and pass the strings to be joined as an argument.

Checking String Properties

Next, let's explore some methods that allow us to check properties of strings. These can be very helpful when you need to validate or categorize string data.

isupper() method

This method checks if all alphabetic characters in the string are uppercase:

```
print("HELLO".isupper()) # Outputs: True
print("Hello".isupper()) # Outputs: False
print("HELLO123".isupper()) # Outputs: True
print("HELLO!".isupper()) # Outputs: True
```

islower() method

This method checks if all alphabetic characters in the string are lowercase:

```
print("hello".islower()) # Outputs: True
print("Hello".islower()) # Outputs: False
print("hello123".islower()) # Outputs: True
print("hello!".islower()) # Outputs: True
```

istitle() method

The `istitle()` method checks if the string is titlecased (i.e., all words start with an uppercase letter followed by lowercase letters):

```
print("Hello World".istitle()) # Outputs: True
print("Hello world".istitle()) # Outputs: False
print("Hello 123".istitle()) # Outputs: True
print("Hello, World!".istitle()) # Outputs: True
```

Note that for all these methods, non-alphabetic characters don't affect the result. The methods only consider alphabetic characters when making their determination.

These methods can be very useful for validating user input or processing text data. For example, you might use `isupper()` to check if a user has entered a code in all caps, or `istitle()` to verify if a name is properly capitalized.

Remember, these methods return True or False, which can be used in more complex logic as you advance in your Python journey.

Replacing Parts of Strings

The `replace()` method allows you to substitute one substring with another.

Basic syntax:

```
new_string = original_string.replace(old_substring,
new_substring)
```

For example:

```
text = "I like apples. Apples are tasty."
new_text = text.replace("apples", "oranges")
print(new_text) # Output: I like oranges. Apples are
# tasty.
```

Notice that only the lowercase "apples" was replaced. The `replace()` method is case-sensitive.

You can also replace multiple occurrences:

```
text = "I like apples. Apples are tasty."
new_text = text.replace("apples",
"oranges").replace("Apples", "Oranges")
print(new_text) # Output: I like oranges. Oranges are
# tasty.
```

String Alignment

Finally, let's explore methods for aligning text:

ljust(): left-justify the string

```
text = "Python"
print(text.ljust(10, '-')) # Output: Python----
```

rjust(): right-justify the string

```
text = "Python"
print(text.rjust(10, '-')) # Output: ----Python
```

c) center(): center the string

```
text = "Python"
print(text.center(10, '-')) # Output: --Python--
```

In each of these methods, the first argument (10 in our examples) specifies the total width of the resulting string. The second argument is the fill character (default is a space).

These alignment methods are great for creating neat, formatted output, such as in tables or columns of text.

Practice these concepts with different strings and see how they work. Don't be afraid to experiment – that's the best way to learn!

zfill() for Padding

The `zfill()` method adds zeros to the left side of a string until it reaches a specified length. What `zfill()` does:

- It pads the left side of a string with zeros.
- If the string is already longer than the specified length, it doesn't change anything.
- For numbers, it respects the sign (+ or -).

Let's see it in action:

```
number = "42"
padded_number = number.zfill(5)
print(padded_number)

# Output: 00042
```

It works with negative numbers too:

```
negative_number = "-42"
padded_negative = negative_number.zfill(5)
print(padded_negative)

# Output -0042
```

Notice how the minus sign stays at the front.

When is this useful? `zfill()` is great for formatting numbers, like making sure all your numbers have the same number of digits. This can be helpful in finance, data processing, or when you need to sort strings that represent numbers.

While `zfill()` works on strings that represent numbers, it's best to use it on strings, not actual number types like integers or floats.

Counting Occurrences

The `count()` method helps you count how many times a substring appears within a larger string.

How to use `count()`:

- Basic syntax: `string.count(substring)`
- It returns the number of non-overlapping occurrences of the substring.

Let's try an example:

```
sentence = "She sells seashells by the seashore."
shell_count = sentence.count("she")
print(shell_count)
```

This will output:

```
2
```

Why 2? Because "she" appears twice: in "She" and in "seashells". `count()` is case-sensitive, so it didn't count "She" with a capital 'S'.

You can also specify where to start and end the search:

```
partial_count = sentence.count("se", 10, 30)
print(partial_count)
```

This will count occurrences of "se" only between the 10th and 30th characters of the string. `count()` is

great for analysing text, like finding out how often a word appears in a paragraph.

Working with Numeric Strings

Sometimes, we need to work with numbers that are stored as strings. Python provides two helpful methods to check if a string contains numeric characters: `isdecimal()` and `isnumeric()`.

The `isdecimal()` method checks if all characters in a string are decimal digits (0-9).

```
number1 = "12345"
print(number1.isdecimal()) # Output: True

number2 = "12.34"
print(number2.isdecimal()) # Output: False
```

The `isnumeric()` method, on the other hand, is more inclusive. It returns True for decimal digits, as well as other numeric characters like fractions or superscript numbers.

```
number3 = "½"
print(number3.isnumeric()) # Output: True

number4 = "2"
print(number4.isnumeric()) # Output: True
```

The main difference between these methods is that `isnumeric()` recognizes a wider range of numeric characters, while `isdecimal()` only recognizes standard decimal digits.

Important note: Be careful when using these methods with user input or data from external sources. They don't handle negative numbers or decimal points, so you might need additional checks for those cases.



Conditionals

Now we're going to take things up a notch and explore control structures, specifically conditional statements. Up to now you've learned about syntax and variables but here we're going to start putting them to good use and learn about making decisions in your code. Your programs will start to become more dynamic from this point.

We'll cover `if` statements, `elif` and `else` clauses, as well as nested conditionals, ternary operators, and `match` statements. We'll also walk through some practical examples and exercises.

Introduction to Conditionals

Imagine your baking a cake following a recipe. Sometimes, the recipe might say, 'If the dough is too dry, add more water.' That's exactly what conditional statements do in programming.

Conditional statements are decision-making structures that allow your code to execute different actions based on whether a condition is true or false. They're like forks in the road, letting our program choose which path to take. Most programming from very basic scripts to advance AI systems, boils down to a series of conditionals statements. So, it's very important that you get a solid understanding of how to use them correctly.

In programming, 'flow control' refers to the order in which individual statements, instructions, or function calls are executed.

Without conditionals, our code would execute from top to bottom, always following the same path. Conditionals allow us to create branches in this path, altering the sequence of execution based on different conditions.

The relationship between conditionals and program flow is fundamental. Conditionals act like traffic directors, guiding the flow of our program based on different conditions and inputs. They allow us to create complex, interactive programs that can handle a variety of scenarios.

As you progress in your Python journey, you'll find conditionals playing a crucial role in more advanced concepts like loops and functions, further enhancing your ability to control program flow.

We have several types of conditional statements at our disposition:

1. The `'if'` statement: This is the simplest form. It says, 'If this condition is true, do this action.'
2. Or the `'if-else'` statement: This adds an alternative. It says, 'If this condition is true, do this; otherwise, do that.'

The if Statement

Let's start by looking at the if statement, the simplest form of conditional in Python. The basic structure looks like this:

```
if condition:
    # code to execute if condition is True
```

Let's break this down:

- We start with the keyword `'if'`.
- This is followed by a condition, which is any expression that evaluates to True or False.
- After the condition, we put a colon (`:`). This tells Python that we're about to define a block of code.
- The next line is indented. This indented block contains the code that will run if the condition is True.

Here's a real example:

```
age = 20

if age >= 18:
    print("You are an adult.")
```

In this case, `'age >= 18'` is our condition. Python evaluates this to True or False. Since 20 is indeed

greater than or equal to 18, this condition is True, so the indented print statement will execute.

Remember, Python will try to interpret any condition as a boolean. For example:

```
if 1: # 1 is considered True
    print("This will print")

if 0: # 0 is considered False
    print("This won't print")

if "hello": # Non-empty strings are considered True
    print("This will print")

if "": # Empty strings are considered False
    print("This won't print")
```

Understanding how Python evaluates conditions to True or False is crucial for writing effective if statements."

Indentation in Python

```
if True:
    print("This is inside the if block")
    print("This is also inside the if block")
print("This is outside the if block")
```

Both indented print statements are part of the if block and will only execute if the condition is True. The last print statement, which isn't indented, is always executed regardless of the condition. Consistent indentation is crucial.

Always be mindful of your indentation. It's not just about style in Python – it's a fundamental part of the language's syntax!

Single-line if Statements

Python also allows us to write if statements on a single line. This can be useful for very simple conditions. Here's the syntax:

```
if condition: statement
```

For example:

```
age = 20
if age >= 18: print("You are an adult.")
```

This does the same thing as our earlier example, but in a more compact form. We can even include an else clause:

```
age = 15
print("Adult" if age >= 18 else "Minor")
```

While these one-liners can be convenient, they should be used sparingly. They're great for simple conditions, but they can make your code harder to read if overused or if the condition or statement is too complex. As a rule of thumb:

- Use single-line if statements for very simple, short conditions.
- If you need to do more than one thing based on the condition, use a full if block.
- If the condition itself is complex, use a full if block for clarity.

Remember, in Python, readability counts. It's often better to write clear, easy-to-understand code than to try to cram everything into one line!

if-else Statements

What if we want our condition to do something in case that the condition is false? That's what the **else** clause is for. The syntax for an if-else statement looks like this:

```
if condition:
    # code to execute if condition is True
else:
    # code to execute if condition is False
```

Let's see an example:

```
age = 15

if age >= 18:
    print("You are an adult.")
else:
    print("You are a minor.")
```

In this case, if the condition (**age >= 18**) is False, the code in the else block will execute. This ensures we have a default action and complete logic coverage – we've accounted for all possible outcomes.

The **if-else** structure introduces the concept of mutual exclusivity. This means that only one of the two blocks (if or else) will execute, never both. This makes our code more efficient and our logic clearer.

```
temperature = 25

if temperature > 30:
    print("It's hot!")
else:
    print("It's not hot.")
```

In this example, it's either hot or not – there's no in-between. This mutual exclusivity is a powerful feature of **if-else** statements, allowing us to make clear, binary decisions in our code

if-elif-else Chains

Sometimes, we can have more than two possible outcomes from our condition. That's where `elif` (short for 'else if') comes in. The `if-elif-else` chain allows us to handle multiple conditions.

```
if condition1:
    # code if condition1 is True
elif condition2:
    # code if condition2 is True
elif condition3:
    # code if condition3 is True
else:
    # code if all conditions are False
```

Let's look at an example:

```
score = 85

if score >= 90:
    print("A grade")
elif score >= 80:
    print("B grade")
elif score >= 70:
    print("C grade")
else:
    print("Below C grade")
```

In this `if-elif-else` chain, Python evaluates each condition in order. As soon as it finds a True condition, it executes that block and then exits the entire chain. This means the order of your conditions is crucial!

Consider this incorrect ordering:

```
score = 95

if score >= 70:
    print("C grade")
elif score >= 80:
    print("B grade")
elif score >= 90:
    print("A grade")
```

Even though the score is 95, this would print "C grade" because that's the first condition that evaluates to True. Always order your conditions from most specific to least specific.

It's also worth noting that the `else` clause is optional. You might use `elif` without `else` when you don't need a catch-all:

```
day = "Monday"

if day == "Saturday":
    print("It's the weekend!")
elif day == "Sunday":
    print("It's the weekend!")

# If it's neither Saturday nor Sunday, nothing will print
```

However, be careful when omitting `else`. Make sure you're not leaving out important cases in your logic.

Nested Conditionals

Sometimes, we need to make decisions within decisions. This is where nested conditionals come in. A nested conditional is simply an `if` statement inside another `if` statement. Here's a basic structure:

```
if outer_condition:
    if inner_condition:
        # code if both conditions are True
    else:
        # code if outer is True but inner is False
else:
    # code if outer condition is False
```

Let's see an example:

```
age = 20
has_license = True

if age >= 18:
    if has_license:
        print("You can drive a car.")
    else:
        print("You need to get a license first.")
else:
    print("You're too young to drive.")
```

This allows for more complex decision-making, but it comes with some caveats. Nested conditionals can quickly become hard to read and maintain if you're not careful. Nested conditionals are appropriate when you have truly hierarchical decisions to make. However, always consider if there's a clearer way to express your logic. Remember, code readability is crucial!

Ternary Operators

Ternary operators provide a concise way to write simple `if-else` statements in a single line. The syntax for a conditional expression is:

```
value_if_true if condition else value_if_false
```

Let's see an example:

```
age = 20
status = "adult" if age >= 18 else "minor"
print(status) # Output: adult
```

This compact form is equivalent to:

```
if age >= 18:
    status = "adult"
else:
    status = "minor"
```

Conditional expressions are great for simple, one-line decisions. However, they should be used carefully.

Here's an appropriate use case:

```
discount = 0.1 if is_member else 0
```

And here's when you might want to avoid it:

```
# Avoid this:
result = (x * 2 + y ** 2) if (x > 0 and y < 10) else (x - y) / 2
```

In the second case, a full if-else statement would be clearer. Use conditional expressions to make your code more concise, but never at the expense of clarity.

Complex Conditions

Often, we need to combine multiple conditions to make a decision. Python provides logical operators `'and'`, `'or'`, and `'not'` for this purpose.

Let's start with the syntax:

```
if condition1 and condition2:
    # Execute if both conditions are True

if condition1 or condition2:
    # Execute if at least one condition is True

if not condition:
    # Execute if condition is False
```

Here's a practical example:

```
age = 25
income = 50000

if age >= 18 and income >= 30000:
    print("Eligible for credit card")
else:
    print("Not eligible for credit card")
```

Python evaluates these operators from left to right, with `'not'` having the highest precedence, then `'and'`, then `'or'`.

To control the order of evaluation or to make your code clearer, use parentheses:

```
if (age >= 18 and income >= 30000) or (credit_score > 700):
    print("Eligible for credit card")
```

Parentheses override the default precedence, ensuring the `'and'` operation is evaluated first.

Remember, while complex conditions can be powerful, they can also make your code harder to read. If a condition becomes too complex, consider

breaking it down into smaller, more manageable parts using variables or functions.

Short-Circuit Evaluation

Short-circuit evaluation is a feature that can make your code more efficient. With short-circuit evaluation, Python stops evaluating as soon as the final result is determined. This applies to the `'and'` and `'or'` operators.

For `'and'` operations:

■ If the first operand is False, Python returns False without evaluating the second operand.

For `'or'` operations:

■ If the first operand is True, Python returns True without evaluating the second operand.

Let's see some examples:

```
name = "Gunnar"

# Using 'and'
result = name.isupper() and len(name) == 6
# Output:
# Checking if name is upper case

# Using 'or'
result = isupper(4) or len(name) == 6
# Output:
# Checking if name is uppercase
# Checking if name is 6 characters long
```

We can leverage this for efficiency:

```
# Efficient:
if x != 0 and y / x > 2:
    print("Condition met")

# Less efficient and potentially error-prone:
if y / x > 2 and x != 0:
    print("Condition met")
```

In the efficient version, we avoid a potential division by zero error by checking `x != 0` first. If `x != 0` evaluates to false, the second operand is not evaluated and doesn't produce an error.

The match Statement

The `match` statement brings structural pattern matching to Python, offering a more expressive and readable way to handle complex conditional logic.

Let's start with the basic syntax:

```
match value:
    case pattern1:
        # code to execute if value matches pattern1
    case pattern2:
```

```
# code to execute if value matches pattern2
case _:
    # default case, like 'else' in if-elif-else
```

Here's a simple example:

```
value = 42

match value:
    case int():
        print("It's an integer")
    case str():
        print("It's a string")
    case list():
        print("It's a list")
    case _:
        print("It's something else")
```

The match statement checks for the type of the value. It does so by determining if the `int()`, `str()`, or `list()` functions evaluate to `True`. As soon as one does, the loop executes the print command and exits the match statement.

Compared to traditional `if-elif-else` chains, the match statement offers several advantages:

1. Readability: Complex conditions become more readable and self-documenting.
2. Expressiveness: It's easier to express complex patterns and structure matching.
3. Performance: In many cases, it can be more efficient than long `if-elif` chains.

While the `match` statement is powerful, it's not always the best choice. For simple conditions, traditional if statements might be more appropriate. As with any feature, use it where it enhances readability and maintainability of your code.



Loops

Loops are a way to repeat a block of code multiple times. Think of them as a way to automate repetitive tasks. Instead of writing the same code over and over again, you can use a loop to do it for you.

For example, imagine you want to print the numbers from 1 to 5. Without loops, you'd have to write:

```
print(1)
print(2)
print(3)
print(4)
print(5)
```

But with loops, you can do this much more efficiently. We'll see how shortly!

Why are Loops Important?

1. They help us process large amounts of data quickly.
2. They're crucial for implementing many algorithms.
3. They reduce code redundancy, making our programs shorter and easier to read.

Python has two main types of loops: `for` loops and `while` loops. So, let's start with the for loop.

The 'for' loop

A `for` loop is used to iterate over a sequence (like a list, string, or range of numbers) called an iterable. Here's the basic syntax (don't worry about iterable – we'll cover that soon):

```
for item in iterable:
    # Code to be repeated
```

The item is a variable that takes on each value in the iterable one by one. The iterable is the sequence you're looping over, for example a string or a list of numbers.

```
for number in range(1, 6):
    print(number)
```

This will print: **1 2 3 4 5**

Let's break down what's happening:

- ▣ range(1, 6) creates a sequence of numbers from 1 to 5 (6 is not included).
- ▣ The loop runs once for each number in this range.
- ▣ Each time, **number** takes on the next value in the range.
- ▣ We print each number.

The range() Function

The **range()** function is often used with for loops. It generates a sequence of numbers.

Basic usage: **range(stop)**

- ▣ Creates numbers from 0 up to, but not including, stop.

Extended usage: **range(start, stop, step)**

- ▣ start: Where to start (default is 0)
- ▣ stop: Where to end (not included)
- ▣ step: Size of increment each time (default is 1)

```
for i in range(0, 10, 2):
    print(i)
```

This will print: 0, 2, 4, 6, 8

The 'while' loop

While loops repeat as long as a condition is True. Here's the basic syntax:

```
while condition:
    # Code to be repeated
```

Example:

```
count = 0
while count < 5:
    print(count)
    count += 1
```

This will print: 0, 1, 2, 3, 4

Here's what's happening:

- ▣ We start with count = 0.
- ▣ The loop continues as long as count < 5 is True.
- ▣ Each time through the loop, we print count and then increase it by 1.
- ▣ When count becomes 5, the condition is False, and the loop stops.

Important: Always make sure the condition will eventually become false, or you'll create an infinite loop!

Loop Control Statements

Sometimes, we need to change how a loop behaves. Python gives us three special statements to do this: **break**, **continue**, and **pass**.

The break statement

The break statement allows us to exit a loop before it has finished all its iterations.

Syntax:

```
while condition:
    # code
    if some_condition:
        break
    # more code
```

For example:

```
for i in range(5):
    if i == 3:
        print("We're breaking the loop!")
        break
    print(i)

print("Loop finished")
```

This will output:

```
0
1
2
We're breaking the loop!
Loop finished
```

When using break in nested loops, it only breaks out of the innermost loop.

The continue statement

The continue statement skips the rest of the current iteration and moves to the next one.

Syntax:

```
while condition:
    # code
    if some_condition:
        continue
    # more code that won't run if continue is triggered
```

Example:

```
for i in range(5):
    if i == 2:
        print("Skipping 2!")
        continue
    print(i)
```

This will output:

```
0
1
Skipping 2!
3
4
```

The pass statement

The pass statement is a placeholder that does nothing. It's useful when you need a statement syntactically but don't want to execute any code.

```
for item in iterable:
    pass
```

Example:

```
for i in range(5):
    if i == 2:
        pass # We'll do something here later
    else:
        print(i)
```

We do this when we're writing the logic of our code, but we don't want to flesh out the individual parts just yet.

The else clause in loops

The else clause in a loop executes when the loop completes normally (without a break).

Syntax:

```
for item in iterable:
    # loop body
else:
    # code to execute if the loop completes normally
for i in range(5):
    print(i)
    if i == 10: # This will never be True
        break
else:
    print("Loop completed normally")
```

This will output all numbers from 0 to 4, followed by "Loop completed normally".

The enumerate() Function

The enumerate() function is useful when you need both the index and the value in a loop.

Syntax:

```
for index, value in enumerate(iterable, start=0):
    # code
```

Here's an example where we keep track of the current iteration index that we print along with the value:

```
fruits = ["apple", "banana", "cherry"]
for index, fruit in enumerate(fruits):
    print(f"Index {index}: {fruit}")
```

This will output:

```
Index 0: apple
Index 1: banana
Index 2: cherry
```

If you need to, you can change the starting index by setting the start parameter:

```
for index, fruit in enumerate(fruits, start=1):
    print(f"Fruit #{index}: {fruit}")
```

Use cases for non-zero start:

- 🟢 When you're dealing with 1-indexed data. For example, some languages, such as MATLAB, start their list index at 1 rather than at 0.
- 🟢 When you want to number items starting from a specific value

Infinite Iterators (advanced)

An iterator is simply an object that we can loop over, like a list of items. But what if we want a list that never ends? That's where infinite iterators come in handy! They're like magical lists that keep going and going, giving us new values whenever we ask for them.

We'll be looking at three types of infinite iterators from the `itertools` module: `count()`, `cycle()`, and `repeat()`. Let's break them down one by one.

itertools.count()

The `count()` iterator is like a number counter that starts from a given number and keeps counting up forever.

Here's how you can use it:

```
from itertools import count

counter = count(start=1)
for number in counter:
    print(number)
    if number == 5:
        break
```

You will see a new command in the script “`from itertools import count`”. This command tells Python that the logic for the count command comes from a module called `itertools`. We will look at this in more detail soon, but for now, it is enough to know that some code will be in different packages and we can import and then use certain parts of these packages using the import command.

In this case we are importing the function “`count`” and we’ll start counting from 1, and the loop will print: 1 2 3 4 5. Notice how we used a break statement to stop the loop. Without it, the counter would go on forever!

A practical use-case for `count()` would be generating unique IDs for items in a database or game objects. Since you are counting up, you always get a unique number.

`itertools.cycle()`

The `cycle()` iterator repeats a sequence of items over and over again. This time we’re importing the function `cycle()` from the `itertools` package.

```
from itertools import cycle

colours = ['red', 'green', 'blue']
colour_cycle = cycle(colours)

for i in range(6):
    print(next(colour_cycle))
```

This will print: red green blue red green blue. As you can see, it keeps cycling through the colours in our list. The `cycle()` function is great for creating alternating patterns or rotating through a set of options.

`itertools.repeat()`

Finally, the `repeat()` iterator, well, repeats a single item over and over again.

```
from itertools import repeat

for message in repeat("Hello!", 3):
    print(message)
```

This will print: Hello! Hello! Hello!

You can also use `repeat()` without specifying a number of repetitions, which will make it repeat indefinitely.

When and How to Use Infinite Iterators

Infinite iterators are super useful when you need a continuous stream of data or when you're not sure how many items you'll need in advance. They're great for:

- Generating unique IDs: Use `count()` to create an endless supply of unique numbers.
- Cyclical data processing: Use `cycle()` when you need to repeatedly process a fixed set of items.
- Default values: Use `repeat()` to fill data structures with a default value.

Common Loop Patterns

In programming, we often need to process lists of data. Python provides several ways to work with lists using loops. Here, we'll cover three main patterns: filtering, transforming, and accumulating.

Filtering Elements

Filtering is when you want to keep only certain items from a list based on a condition. Let's say we have a list of numbers and we want to keep only the even ones:

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_numbers = []

for num in numbers:
    if num % 2 == 0:
        even_numbers.append(num)

print(even_numbers) # Output: [2, 4, 6, 8, 10]
```

Here, we use an `if` statement inside the loop to check if each number is even. If it is, we add it to our new list. The `append()` method adds a new element to an existing list.

Transforming Elements

Sometimes we want to apply a change to every item in a list. This is called transforming. Let's double every number in our list:

```
numbers = [1, 2, 3, 4, 5]
doubled_numbers = []

for num in numbers:
    doubled_numbers.append(num * 2)

print(doubled_numbers) # Output: [2, 4, 6, 8, 10]
```

Accumulating Results

Accumulating is when we want to combine all the items in a list into a single result. A common example is summing up numbers:

```
numbers = [1, 2, 3, 4, 5]
total = 0

for num in numbers:
    total += num

print(total) # Output: 15
```

Here, we use a "running total" variable that we update in each iteration of the loop.

Note: Python has built-in functions like `sum()` that can do this for you, but it's important to understand the concept of accumulation.

Nested Loops

Nested loops are loops inside other loops. They're useful when you need to work with multi-dimensional data or when you need to compare each item in a list with every other item.

Here's a simple example that prints a 5 x 5 multiplication table:

```
for i in range(1, 6):
    for j in range(1, 6):
        print(i * j, end="\t")
    print() # This print() is to start a new line after
            # each row
```

Output:

1	2	3	4	5
2	4	6	8	10
3	6	9	12	15
4	8	12	16	20
5	10	15	20	25

In this example, the outer loop (with variable `i`) represents the rows, and the inner loop (with variable `j`) represents the columns. For each value of `i`, the inner loop runs completely before moving to the next value of `i`.

Nested loops are great for working with 2D data like tables or matrices. For example, if you had a list of lists

representing a grid, you could use nested loops to process each cell:

```
grid = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

for row in grid:
    for cell in row:
        print(cell, end=" ")
    print() # New line after each row
```

Output:

```
1 2 3
4 5 6
7 8 9
```

The `iter()` and `next()` Functions

You will often need to work with collections of data, like lists or strings. Iterators provide a way to access elements in these collections one at a time, without loading everything into memory at once. This is especially useful when dealing with large datasets.

The iterator protocol is a set of rules that objects follow to become iterable. An iterable is any object that can be looped over, like a list. Python provides two built-in functions to work with iterators: `iter()` and `next()`.

- 🟢 `iter()` creates an iterator object from an iterable.
- 🟢 `next()` retrieves the next item from the iterator.

Let's look at an example to see this in action:

```
my_list = [1, 2, 3, 4, 5]
my_iterator = iter(my_list)

print(next(my_iterator)) # Output: 1
print(next(my_iterator)) # Output: 2
print(next(my_iterator)) # Output: 3
```

When an iterator has no more items to return, it raises a `StopIteration` exception. This tells Python that we've reached the end of the iterator.

```
# Continuing from the previous example
print(next(my_iterator)) # Output: 4
print(next(my_iterator)) # Output: 5
print(next(my_iterator)) # Raises StopIteration
```

The concepts in this chapter will help you write more efficient and safer Python code. Remember to practice with your own examples to reinforce your learning.



Lists

Lists are like containers that can hold multiple items. Imagine a shopping list where you write down all the things you need to buy. In Python, a list works similarly, but instead of groceries, it can hold any type of data - numbers, text, or even other lists!

They are an ordered collection of items. This means the items stay in the order you put them in.

Lists are super helpful when you need to work with groups of related data. For example, you might use a list to store the names of all your friends, the scores in a game, or the items in your digital shopping cart.

Creating Lists

Sometimes, you might want to start with an empty list and add items to it later. Here's how you can create an empty list:

```
my_empty_list = []
```

It's that simple! Just a pair of square brackets. More often, you'll want to create a list with some items already in it. Here's how:

```
fruits = ["apple", "banana", "cherry"]
numbers = [1, 2, 3, 4, 5]
mixed_list = [42, "hello", True, 3.14]
```

As you can see, we can put different types of data in a list - numbers, strings, even boolean values!

Python provides another way to create lists using the `list()` constructor, which is similar to the `str()`,

`int()`, and `float()` functions that we've seen before when we looked at variables. This is particularly useful when you want to convert other types of data into a list:

```
# Creating a list from a string
letter_list = list("Python")
print(letter_list) # ['P', 'y', 't', 'h', 'o', 'n']
# Creating a list from a range of numbers
number_list = list(range(1, 6))
print(number_list) # [1, 2, 3, 4, 5]
```

The `list()` constructor can turn other "iterable" objects into lists. Remember that strings and ranges are examples of things you can convert to lists this way.

List Elements

One of the coolest things about Python lists is that they can contain different types of data all in one list:

```
my_list = [42, "hello", 3.14, True]
```

In this list, we have an integer (42), a string ("hello"), a float (3.14), and a boolean (True). This flexibility allows you to store related information together, even if it's not all the same type.

Lists can also contain other lists! This is called nesting, and it's really useful for creating more complex data structures. Here's an example:

```
nested_list = [1, 2, [3, 4, 5], 6]
```

In this list, the third element `[3, 4, 5]` is itself a list. You can even have lists within lists within lists:

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Accessing List Elements

Now that we know what can go in a list, let's talk about how to get things out of a list.

We use indexing to access individual elements in a list just like we did with strings. Here's the key thing to remember: indexing starts at 0, not 1.

```
my_list = ["apple", "banana", "cherry", "date"]
```

To access elements, we use square brackets `[]` after the list name:

```
print(my_list[0]) # This will print "apple"
print(my_list[2]) # This will print "cherry"
```

Just like with strings, we can also use negative indexing, which counts from the end of the list. The last element is at index -1, the second-to-last at -2, and so on.

```
print(my_list[-1]) # This will print "date"
print(my_list[-3]) # This will print "banana"
```

This is really handy when you want to access elements from the end of the list without knowing how long the list is.

Be careful not to use an index that's outside the range of your list though. If your list has 4 elements, trying to access `my_list[4]` or `my_list[-5]` will result in an error.

Being able to store different types of variables is highly useful. Imagine you're making a game where players can collect different types of items. You could use a list to store each player's inventory:

```
player_inventory = ["sword", 5, "shield", True, ["health potion", "mana potion"]]
```

This list contains the player's weapon, the amount of gold they have, their armour, whether they have a special ability, and a nested list of potions.

You could then access these elements like this:

```
print("Player weapon:", player_inventory[0])
print("Player gold:", player_inventory[1])
print("First potion:", player_inventory[-1][0])
```

The use of lists is truly limitless as they can be used to organize and easily access different types of related data in your programs.

List Slicing

List slicing is like cutting a slice of pizza - you're taking a portion of your list. It's a powerful way to work with specific parts of your data.

The basic syntax for slicing is `[start:end]`. **Start** is the index where you want to begin (inclusive) and **end** is the index where you want to stop (exclusive)

```
fruits = ["apple", "banana", "cherry", "date", "elderberry"]
print(fruits[1:4]) # Output: ['banana', 'cherry', 'date']
```

In this example, we're getting a slice from index 1 to index 4 (but not including 4).

You can also add a **step** to your slice, which determines the interval between elements.

```
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print(numbers[1:8:2]) # Output: [1, 3, 5, 7]
```

Here, we're starting at index 1, ending at index 8, and taking every 2nd element. Just as with strings, you can even use a negative step to reverse the order:

```
print(numbers[7:2:-1]) # Output: [7, 6, 5, 4, 3]
```

Python allows you to omit some or all of the slice parameters.

```
print(numbers[:5]) # From start to index 5 (excl.)
print(numbers[5:]) # From index 5 to end
print(numbers[:]) # The entire list
print(numbers[::-1]) # The entire list in reverse order
```

Modifying Lists

Now that we know how to slice lists, let's learn how to modify them. Since lists are mutable, you can change a single element by assigning a new value to a specific index:

```
fruits = ["apple", "banana", "cherry"]
fruits[1] = "blueberry"
print(fruits) # Output: ['apple', 'blueberry', 'cherry']
```

You can also modify multiple elements at once using slicing.

```
numbers = [0, 1, 2, 3, 4, 5]
numbers[2:5] = [20, 30, 40]
print(numbers) # Output: [0, 1, 20, 30, 40, 5]
```

You can even replace a slice with a sequence of a different length.

```
letters = ['a', 'b', 'c', 'd', 'e']
letters[1:4] = ['X', 'Y']
print(letters) # Output: ['a', 'X', 'Y', 'e']
```

List Methods

Let's dive into the different list methods, see how they work, and explore some practical uses for each. We'll look at most list methods one by one, but there are plenty more for you to look up in the Python documentation.

 **append()**: Adding elements to the end

The `append()` method adds a single element to the end of a list.

```
fruits = ["apple", "banana"]
fruits.append("cherry")
print(fruits) # Output: ["apple", "banana", "cherry"]
```

Note that `append()` modifies the list directly (in-place), so you don't need to assign the result to a new variable. Appending to lists is useful when you are for example building a list of user inputs or accumulating results in a loop.

■ `extend()`: Adding multiple elements

The `extend()` method adds all elements from another iterable to the end of the current list. It's worth noting that `extend()` is more efficient than using `append()` in a loop when adding multiple elements.

```
fruits = ["apple", "banana"]
more_fruits = ["cherry", "date"]
fruits.extend(more_fruits)
print(fruits) # Output: ["apple", "banana", "cherry", "date"]
```

■ `insert()`: Adding elements at specific positions

The `insert()` method adds an element at a specific position: `list.insert(index, element)`. Existing elements are shifted to make room for the new element.

```
fruits = ["apple", "banana", "cherry"]
fruits.insert(1, "blueberry")
print(fruits) # Output: ["apple", "blueberry", "banana", "cherry"]
```

■ `remove()`: Removing specific elements

The `remove()` method removes the first occurrence of a specified value from the list. If the element isn't in the list, `remove()` will raise a `ValueError`. It's often safe to use it in a try-except block – something that we'll cover in a few chapters' time.

```
fruits = ["apple", "banana", "cherry", "banana"]
fruits.remove("banana")
print(fruits) # Output: ["apple", "cherry", "banana"]

try:
    fruits.remove("grape")
except ValueError:
    print("Grape not found in the list")
```

■ `pop()`: Removing and returning elements

The `pop()` method removes and returns an element at a specified index. If no index is provided, it removes and returns the last element.

```
fruits = ["apple", "banana", "cherry"]
last_fruit = fruits.pop()
print(last_fruit) # Output: "cherry"
print(fruits) # Output: ["apple", "banana"]

first_fruit = fruits.pop(0)
print(first_fruit) # Output: "apple"
print(fruits) # Output: ["banana"]
```

■ `clear()`: Removing all elements

The `clear()` method removes all elements from the list, leaving it empty.

```
fruits = ["apple", "banana", "cherry"]
fruits.clear()
print(fruits) # Output: []
```

■ `index()`: Finding the index of an element

You have seen this method before when looking at strings. The `index()` method returns the index of the first occurrence of a specified element: `list.index(element, start, end)`. If the element isn't found, `index()` raises a `ValueError`. Use it in a try-except block for safety.

```
fruits = ["apple", "banana", "cherry", "date", "banana"]
banana_index = fruits.index("banana")
print(banana_index) # Output: 1

# Using start and end parameters
late_banana = fruits.index("banana", 2)
print(late_banana) # Output: 4
```

■ `count()`: Counting occurrences of an element

The `count()` method returns the number of times a specified element appears in the list.

```
fruits = ["apple", "banana", "cherry", "banana"]
banana_count = fruits.count("banana")
print(banana_count) # Output: 2
```

■ `sort()`: Sorting the list in-place

The `sort()` method sorts the list in ascending order by default. You can tell sort to sort in descending order by using the parameter `reverse = True`. For more complex sorting, you can use the key parameter, but that's a more advanced topic

```
numbers = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]
numbers.sort()
print(numbers) # Output: [1, 1, 2, 3, 3, 4, 5, 5, 5, 6, 9]

numbers.sort(reverse=True)
print(numbers) # Output: [9, 6, 5, 5, 5, 4, 3, 3, 2, 1, 1]
```

■ `reverse()`: Reversing the list in-place

The `reverse()` method, you guessed it, inverts the order of elements in the list.

```
fruits = ["apple", "banana", "cherry"]
fruits.reverse()
print(fruits) # Output: ["cherry", "banana", "apple"]
```

■ `copy()`: Creating a shallow copy

The `copy()` method creates a new list with the same elements as the original list. A shallow copy, means it creates a new list but doesn't create copies of the elements themselves. This distinction becomes important when working with lists of complex objects, but for now, just remember that `copy()` creates a new, independent list.

```
original = ["apple", "banana", "cherry"]
copied = original.copy()
copied.append("date")
print(original) # Output: ["apple", "banana", "cherry"]
print(copied)   # Output: ["apple", "banana", "cherry", "date"]
```

These list methods provide powerful tools for manipulating and analysing data in Python. As you work with lists, you'll find yourself using these methods frequently to solve various programming challenges. Practice using these methods in your own code to become more comfortable with them.

Remember, most of these methods modify the list in-place, meaning they change the original list rather than creating a new one. This can be very efficient, but it's important to be aware of when you're modifying your original data.

List Operations

Concatenation (+)

In Python, we can combine two or more lists using the `+` operator. This is called concatenation.

```
fruits = ['apple', 'banana']
vegetables = ['carrot', 'broccoli']
grocery_list = fruits + vegetables
print(grocery_list) # Output: ['apple', 'banana', 'carrot', 'broccoli']
```

Note that concatenation creates a new list without modifying the original lists.

Repetition (*)

We can also create a new list by repeating an existing list multiple times using the `*` operator.

```
cheers = ['hip', 'hop']
longer_cheers = cheers * 3
print(longer_cheers) # Output: ['hip', 'hop', 'hip', 'hop', 'hip', 'hop']
```

This is useful when you need to quickly create a list with repeated elements.

List Length

To find out how many elements are in a list, we use the `len()` function.

```
numbers = [1, 2, 3, 4, 5]
list_length = len(numbers)
print(list_length) # Output: 5
```

Important note: While list indices start at 0, `len()` gives us the actual count of elements. So, a list with 5 elements will have indices 0 to 4, but `len()` will return 5.

Checking Membership

We can check if an element exists in a list using the `in` operator. This returns a boolean value (True or False).

```
fruits = ['apple', 'banana', 'cherry']
print('banana' in fruits) # Output: True
print('mango' in fruits)  # Output: False
```

This is particularly useful in conditional statements:

```
if 'apple' in fruits:
    print("We have apples!")
```

List Comprehensions

Next, we're going to explore a powerful and concise way to create lists. If you've been using loops to build lists, you're in for a treat. List comprehensions can often replace those loops with a single line of code that's both readable and efficient. The more you get into Python, the more you will use list comprehensions to clean up your code.

Simple List Creation with Comprehensions

Let's start with the basic syntax of a list comprehension:

```
[expression for item in iterable]
```

This might look a bit strange at first, but let's break it down:

- The square brackets `[]` tell Python we're creating a list.
- `expression` is what we want to put into our new list.
- `for item in iterable` is like a mini for-loop inside the brackets.

```
# Let's create a list of squares of numbers from 0 to 4
squares = [x**2 for x in range(5)]
print(squares) # Output: [0, 1, 4, 9, 16]
```

In this example, `x**2` is our expression, and `range(5)` is our iterable. We're saying "for each number `x` in the range 0 to 4, put `x` squared into our new list. This is much more concise than using a traditional for loop:

```
squares = []
for x in range(5):
    squares.append(x**2)
```

Adding Conditions to List Comprehensions

We can make our list comprehensions even more powerful by adding conditions:

```
[expression for item in iterable if condition]
```

Let's see an example:

```
# Create a list of even squares from 0 to 9
even_squares = [x**2 for x in range(10) if x % 2 == 0]
print(even_squares) # Output: [0, 4, 16, 36, 64]
```

Here, we're only including the square if the original number is even – `%` gives the remainder from a division operation.

List comprehensions are often more readable for simple operations, but be careful! If your logic gets too complex, a traditional for loop might be clearer. Always prioritize readability. As for performance, list comprehensions are generally faster for creating lists, especially for larger datasets. However, the difference is usually small for simple operations, so choose based on readability first.

Nested List Comprehensions

We can also create nested list comprehensions. These are useful for working with nested lists or creating 2D lists:

```
# Create a 3x3 matrix of multiplication table
matrix = [[i*j for j in range(1, 4)] for i in range(1, 4)]
print(matrix) # Output: [[1, 2, 3], [2, 4, 6], [3, 6, 9]]
```

This creates a list of lists. The outer comprehension creates three inner lists, and each inner comprehension fills those lists with values. This might sound a little complicated and that's because it can take a moment to get your head around this. Try to understand nested list comprehensions before

moving on. We can also use nested comprehensions to flatten nested structures:

```
# Flatten a list of lists
nested = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
flattened = [num for sublist in nested for num in
              sublist]
print(flattened) # Output: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

This reads as "for each sublist in nested, for each num in sublist, include num in our new list".

As a rule of thumb, while list comprehensions are powerful, don't overuse them. If your comprehension becomes hard to read at a glance, consider using a regular for loop instead. Most importantly, readability counts! It's better to have clear, understandable code than to squeeze everything into a single line.

Copying Lists

Sometimes, you'll want to create a copy of a list instead of just referring to the original. There are two main ways to copy a list: shallow copy and deep copy.

■ A shallow copy creates a new list object, but the elements inside are references to the same objects as in the original list.

```
original = [1, 2, [3, 4]]
shallow_copy = original[:] # Using slice notation for
                           # shallow copy

print(original)
print(shallow_copy)

# Modifying the nested list
original[2][0] = 5

print(original)
print(shallow_copy) # The nested list is also modified
```

In this example, `original[:]` creates a shallow copy. It's a quick way to create a new list object with the same top-level elements.

■ A deep copy creates a completely independent copy of the list and all nested objects.

To create a deep copy, we need to import the `copy` module:

```
import copy

original = [1, 2, [3, 4]]
deep_copy = copy.deepcopy(original)

print(original)
print(deep_copy)

# Modifying the nested list
original[2][0] = 5

print(original)
print(deep_copy) # The nested list remains unchanged
```

When to use each:

- 🟢 Use a shallow copy when you only need to modify the top-level list.
- 🟢 Use a deep copy when you need a completely independent copy, especially with nested structures.

List Unpacking

List unpacking allows you to assign multiple variables at once from a list.

```
a, b, c = [1, 2, 3]
print(a, b, c) # Output: 1 2 3
```

This is especially useful when you know the exact structure of your list. What happens when you don't know the length? If you are handling mismatched lengths you might get a `ValueError` as shown in the code. You can use the `*` operator to catch any number of remaining values in a list. That way you avoid any `ValueError`.

```
# Too few values
a, b, c = [1, 2] # This will raise a ValueError

# Too many values
a, b, c = [1, 2, 3, 4] # This will also raise a
                      # ValueError

# Using * to catch extra values
a, b, *rest = [1, 2, 3, 4, 5]
print(a, b, rest) # Output: 1 2 [3, 4, 5]
```

The None Object in Lists

`None` is a special object in Python that represents the absence of a value. It's often used as a placeholder in lists. Always use `is` or `is not` when comparing with `None`, `not ==` or `!=`.

```
values = [1, None, 3, None, 5]

# Checking for None
for value in values:
    if value is None:
        print("Found a None value")
    else:
        print(f"Found a number: {value}")
```

Removing Duplicates from a List

Sometimes you want a list with no repeated elements. Here's how to do it while keeping the original order:

```
numbers = [1, 2, 2, 3, 4, 3, 5]
unique_numbers = []

for num in numbers:
    if num not in unique_numbers:
        unique_numbers.append(num)
```

```
print(unique_numbers) # Output: [1, 2, 3, 4, 5]
```

This method goes through each element and only adds it to the new list if it's not already there.

Remember, this method keeps the order of elements as they first appear in the original list.

Lists in String Formatting

You already know how to create strings and lists. Now, let's combine them! Python gives us a neat way to include list elements in our strings using f-strings.

```
fruits = ["apple", "banana", "cherry"]
print(f"I love {fruits[0]} and {fruits[2]}!")
```

This will print: "I love apple and cherry!"

We can access list elements directly inside the curly braces of an f-string. This is super handy when you want to include specific list items in your output.

You can also format list elements within the string:

```
prices = [3.99, 2.50, 1.75]
print(f"The first item costs ${prices[0]:.2f}")
```

This will print: "The first item costs \$3.99"

The `:.2f` part tells Python to show the number with two decimal places.

This is great for creating user-friendly output in your programs, like displaying shopping lists or game inventories.

Comparing Lists

Python allows us to compare entire lists using comparison operators like `==`, `!=`, `<`, `>`, `<=`, and `>=`. This can be really useful!

```
list1 = [1, 2, 3]
list2 = [1, 2, 3]
list3 = [1, 2, 4]

print(list1 == list2) # True
print(list1 == list3) # False
print(list1 < list3) # True
```

When comparing lists, Python checks each element in order:

- 🟢 `==` checks if all elements are the same
- 🟢 `<` checks if list1's elements are less than list2's elements

If the lists have different lengths, Python compares until it reaches the end of the shorter list.

```
short_list = [1, 2]
long_list = [1, 2, 3]
print(short_list < long_list) # True
```

The del Statement with Lists

Sometimes, you need to remove items from a list. The 'del' statement is a powerful tool for this.

To delete a single element:

```
numbers = [1, 2, 3, 4, 5]
del numbers[2]
print(numbers) # [1, 2, 4, 5]
```

To delete a slice (multiple elements):

```
letters = ['a', 'b', 'c', 'd', 'e']
del letters[1:4]
print(letters) # ['a', 'e']
```

You can even delete the entire list:

```
my_list = [1, 2, 3]
del my_list
# my_list no longer exists!
```

That wraps up our exploration of list operations. Remember, practice makes perfect! Try using these concepts in your own programs.!



Tuples

A tuple is a way to group multiple items together in Python. Think of it like a container that can hold different pieces of information.

We create tuples using parentheses (). Items are separated by commas and they're immutable, which means that once you create a tuple, you can't change its contents. Here is a simple example:

```
my_tuple = (1, 2, 3, "apple", "banana")
print(my_tuple)
```

This will output: `(1, 2, 3, 'apple', 'banana')`

Tuples maintain the order of items. This means the first item you put in will always be the first item, the second will always be second, and so on.

Now, you might be wondering: "Why use tuples if we already have lists?" Great question! Let's compare them:

- ✔ Use tuples for collections that shouldn't change (like days of the week).
- ✔ Use lists for collections that might need updating (like a to-do list).

Tuples are also faster than lists for Python to process and they can be used as keys in dictionaries (which we'll learn about soon).

Creating Tuples

An empty tuple is the simplest form of a tuple. It's created using empty parentheses:

```
empty_tuple = ()
print(empty_tuple) # Output: ()
```

This is different from an empty list, which uses square brackets. Most of the time, you'll initiate tuples with predefined values though.

```
fruits = ("apple", "banana", "cherry")
print(fruits) # Output: ('apple', 'banana', 'cherry')
```

Interestingly, in many cases, you can omit the parentheses:

```
colours = "red", "green", "blue"
print(colours) # Output: ('red', 'green', 'blue')
```

Both fruits and colours are valid tuples. Here's a tricky one for you! To create a tuple with just one element, you need to include a trailing comma:

```
single_fruit = ("apple",)
print(single_fruit) # Output: ('apple',)
```

The comma is crucial here. Without it, Python treats it as a regular value in parentheses, not a tuple:

```
not_a_tuple = ("apple")
print(not_a_tuple) # Output: apple
print(type(not_a_tuple)) # Output: <class 'str'>
```

You can also create tuples from other iterable objects using the `tuple()` constructor.

```
list_to_tuple = tuple([1, 2, 3])
print(list_to_tuple) # Output: (1, 2, 3)

string_to_tuple = tuple("hello")
print(string_to_tuple) # Output: ('h', 'e', 'l', 'l', 'o')
```

Tuple Packing and Unpacking

Now, let's look at some cool features of tuples: packing and unpacking. Tuple packing is when Python automatically creates a tuple from a sequence of values:

```
packed_tuple = 1, 2, 3
print(packed_tuple) # Output: (1, 2, 3)
```

Tuple unpacking on the other hand allows you to assign the elements of a tuple to individual variables:

```
x, y, z = (1, 2, 3)
print(x) # Output: 1
print(y) # Output: 2
print(z) # Output: 3
```

The number of variables must match the number of elements in the tuple, or you'll get an error. Alternatively, you can use the `*rest` method that we've seen with lists.

Tuples make it easy to assign multiple values in one line. We have seen how we can swap values of two variables at the beginning of the course. Now you see that it's thanks to tuples underlying the assignment.

```
# Swapping variables
a, b = 5, 10
print(a, b) # Output: 5 10

a, b = b, a
print(a, b) # Output: 10 5
```

In geospatial applications, tuples are great for representing fixed collections of items, like coordinates:

```
point = (3, 4)
x, y = point
print(f"X coordinate: {x}, Y coordinate: {y}")
# Output: X coordinate: 3, Y coordinate: 4
```

Accessing Tuple Elements

To access elements in a tuple, we use good old indexing. Each item in a tuple has a position, or index, that we can use to retrieve it. We can use positive as well as negative indexing, just as we've seen with strings and lists.

```
my_tuple = ("apple", "banana", "cherry")
print(my_tuple[0]) # This will print: apple
print(my_tuple[1]) # This will print: banana
print(my_tuple[-1]) # This will print: cherry
print(my_tuple[-2]) # This will print: banana
```

If we try to access an index that doesn't exist, Python will raise an `IndexError`. This happens when we go beyond the tuple's range.

```
# This will cause an IndexError:
# print(my_tuple[3])
```

To prevent this, always make sure your index is within the tuple's range. You can use the `len()` function to check the tuple's length:

```
if 3 < len(my_tuple):
    print(my_tuple[3])
else:
    print("Index out of range")
```

Tuple Slicing

We can also slice tuples as we've learned before. The basic syntax for slicing is the same as for lists: `[start:end]`. Remember, the start index is inclusive (included), but the end index is exclusive (not included).

```
numbers = (0, 1, 2, 3, 4, 5)
print(numbers[1:4]) # This will print: (1, 2, 3)
```

As we saw before, we can add a third parameter to our slice to specify the step, or interval: `[start:end:step]`

```
print(numbers[0:6:2]) # This will print: (0, 2, 4)
```

We can also use a negative step to reverse the order.

```
print(numbers[::-1]) # This will print (5, 4, 3, 2, 1, 0)
```

A quick refresher: If we omit the start or end index, Python uses default values. The default start is 0, and the default end is the length of the tuple.

```
print(numbers[:3]) # From start to index 3
# (exclusive): (0, 1, 2)
print(numbers[3:]) # From index 3 to end: (3, 4, 5)
print(numbers[:]) # The entire tuple:(0, 1, 2, 3, 4, 5)
```

Immutability of Tuples

Once you create a tuple, you can't modify its elements. This is different from lists, which allow you to change their contents.

```
my_tuple = (1, 2, 3)
print(my_tuple) # Output: (1, 2, 3)

# Trying to change an element
my_tuple[0] = 5 # This will raise a TypeError
```

If you run this code, Python will show an error message like: `TypeError: 'tuple' object does not support item assignment`

This error occurs because tuples are designed to be unchangeable. But why would we want this? Well, immutability has some advantages:

- ▣ Data integrity: You can be sure the data won't accidentally change.
- ▣ Security: Immutable objects can be shared without worrying about modifications.
- ▣ Performance: In some cases, Python can optimize immutable objects for better speed.

Now, you might be thinking, "What if I really need to change a tuple?" While you can't modify a tuple directly, there are workarounds. You can, for example, create a new tuple with the desired changes:

```
my_tuple = (1, 2, 3)
new_tuple = (5,) + my_tuple[1:]
print(new_tuple) # Output: (5, 2, 3)
```

Alternatively, you can convert it to a list, modify, then convert back to a tuple:

```
my_tuple = (1, 2, 3)
my_list = list(my_tuple)
my_list[0] = 5
new_tuple = tuple(my_list)
print(new_tuple) # Output: (5, 2, 3)
```

However, these methods create new tuples rather than modifying the original one.

Tuple Methods

Although tuples are immutable, they do have a couple of useful methods: `count()` and `index()`.

`count()`, no surprise, counts how many times an element appears in the tuple.

```
my_tuple = (1, 2, 2, 3, 2, 4)
print(my_tuple.count(2)) # Output: 3
print(my_tuple.count(5)) # Output: 0 (5 is not in the tuple)
```

As you can see, `count()` returns 0 if the element isn't in the tuple.

`index()` finds the first position (index) of an element in the tuple.

```
my_tuple = (1, 2, 3, 4, 2)
print(my_tuple.index(2)) # Output: 1 (first occurrence of 2)
```

You can also specify a start and end position to search within:

```
print(my_tuple.index(2, 2)) # Output: 4 (searching from index 2 onwards)
```

Be careful though! If the element isn't in the tuple (or in the specified range), Python will raise a `ValueError`. To avoid this error, you can check if the element is `in` the tuple first:

```
element = 5
if element in my_tuple:
    print(my_tuple.index(element))
else:
    print(f"{element} is not in the tuple")
```

Tuple Operations

We can combine two or more tuples using the plus sign (+). This is called concatenation. It creates a new tuple containing all the elements from the tuples we're combining. Let's try it out:

```
fruits = ("apple", "banana")
vegetables = ("carrot", "potato")

grocery_list = fruits + vegetables

print(grocery_list)
```

If you run this code, you'll see:

```
('apple', 'banana', 'carrot', 'potato')
```

As you can see, we've created a new tuple grocery list that contains all the elements from both fruits and vegetables.

We can also repeat a tuple multiple times using the asterisk (*) operator. This creates a new tuple with the original elements repeated.

```
colours = ("red", "blue")
rainbow = colours * 3

print(rainbow)
```

This will output:

```
('red', 'blue', 'red', 'blue', 'red', 'blue')
```

The rainbow tuple now contains the elements of colours repeated three times. Not much of a rainbow, I know.

Using the `len()` function we can find out how many elements are in a tuple. This is super helpful when you need to know the size of your tuple.

Sometimes, you need to check if a specific element exists in your tuple. We easily find out with the `'in'` operator.

```
fruits = ("apple", "banana", "cherry")

if "banana" in fruits:
    print("Yes, we have bananas!")

if "orange" in fruits:
    print("We have oranges too!")
else:
    print("Sorry, no oranges.")
```

This code will output:

```
Yes, we have bananas!
Sorry, no oranges.
```

The `'in'` operator returns a Boolean value (True or False), which makes it perfect for use in conditional statements.

Iterating Over Tuples

You already know about loops, right? Well, guess what - we can use loops with tuples too! This is super handy when we want to go through each item in a tuple one by one.

Let's say we have a tuple of fruits:

```
fruits = ("apple", "banana", "cherry")
```

To go through each fruit, we can use a for loop like this:

```
for fruit in fruits:
    print(fruit)
```

This will print:

```
apple
banana
cherry
```

It's that simple! The loop goes through each item in the tuple, and we can do whatever we want with each item - in this case, we're just printing it.

If you need to know the position (or index) of each item as well, you can use the `enumerate()` function:

```
for index, fruit in enumerate(fruits):
    print(f"Fruit {index + 1}: {fruit}")
```

This will print:

```
Fruit 1: apple
Fruit 2: banana
Fruit 3: cherry
```

Tuples vs. Lists

Now, you might be wondering, "Why use tuples when we have lists?" Great question! Tuples and lists are similar, but they have some key differences that make each useful in different situations.

Tuples are generally faster than lists for reading operations. This is because tuples are immutable - once created. This allows Python to optimize how it handles tuples.

Tuples also use less memory than lists. If you have data that won't change, using a tuple instead of a list can make your program more efficient.

- ✔ Use tuples for data that shouldn't change, like the days of the week or the names of months.
- ✔ Use lists when you need to add, remove, or modify items.

```
# Tuple for days of the week (doesn't change)
days = ("Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday")

# List for your to-do items (changes often)
todo_list = ["Buy groceries", "Do laundry", "Call mom"]

# You can add to the list
todo_list.append("Clean room")

# But you can't add to the tuple
# This would cause an error:
# days.append("Funday")
```

One cool thing about tuples: you can use them as dictionary keys, which you can't do with lists!

Converting Between Tuples and Lists

Sometimes, you might need to switch between tuples and lists. Python makes this super easy with the `list()` and `tuple()` functions.

To convert a tuple to a list:

```
my_tuple = (1, 2, 3)
my_list = list(my_tuple)
print(my_list) # Output: [1, 2, 3]
```

And to convert a list to a tuple:

```
my_list = [4, 5, 6]
my_tuple = tuple(my_list)
print(my_tuple) # Output: (4, 5, 6)
```

Why would you want to do this? Well, lists are mutable (changeable), while tuples are immutable (unchangeable). If you need to modify the contents, you'd convert to a list, as we've seen. If you want to make sure the data doesn't change, you'd use a tuple.

Tuple Comprehensions

Tuple comprehensions, also known as generator expressions, are a way to create tuples quickly based on some existing data. They're similar to list comprehensions, which we have seen before.

Here's the basic syntax:

```
(expression for item in iterable)
```

- The parentheses () indicate we're working with a tuple.
- expression is what we want to put in our tuple.
- item is a variable that represents each element in our iterable.
- iterable is any sequence we can loop over, like a list or another tuple.

```
numbers = (1, 2, 3, 4, 5)
squared_numbers = (n**2 for n in numbers)
```

Now, here's where tuple comprehensions differ from list comprehensions: when you run this code, `squared_numbers` isn't actually a tuple yet. It's something called a generator object. To turn it into a tuple, you need to use the tuple constructor:

```
squared_numbers_tuple = tuple(squared_numbers)
print(squared_numbers_tuple) # Output: (1, 4, 9, 16, 25)
```

This behaviour can be useful when you're working with large amounts of data because it saves memory. The generator doesn't create all the values at once - it generates them as you need them.

Be careful not to confuse tuple comprehensions with list comprehensions. List comprehensions use square brackets [] and create a full list immediately.

The 'zip()' Function

The built-in `zip()` function is great for working with multiple iterables at once. It creates tuples where each tuple contains the i-th element from each of the input iterables.

Here's the basic syntax:

```
zip(iterable1, iterable2, ...)
```

Let's see it in action:

```
names = ["Alice", "Bob", "Charlie"]
ages = [25, 30, 35]

for person in zip(names, ages):
    print(person)

# Output:
# ('Alice', 25)
# ('Bob', 30)
# ('Charlie', 35)
```

`zip()` is really useful when you want to pair up elements from different lists. For example, if you have separate lists of keys and values, you can use `zip()` to create a dictionary:

```
keys = ["name", "age", "city"]
values = ["Alice", 30, "New York"]

person_dict = dict(zip(keys, values))
print(person_dict) # Output: {'name': 'Alice', 'age': 30, 'city': 'New York'}
```

Remember, like tuple comprehensions, `zip()` creates a generator object. If you need an actual list of tuples, you can convert it using a constructor:

```
zipped_list = list(zip(names, ages))
```

Tuples in List Comprehensions

List comprehensions are a powerful feature in Python that allow us to create lists in a concise way. But did you know we can use tuples in list comprehensions too? Let's see how!

Imagine we want to create a list of coordinates. We can use tuples in a list comprehension like this:

```
coordinates = [(x, y) for x in range(3) for y in range(3)]
print(coordinates)
```

This will output:

```
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0),
(2, 1), (2, 2)]
```

Each pair of (x, y) is a tuple representing a coordinate.

We can also create lists of tuples based on existing data:

```
names = ["Alice", "Bob", "Charlie"]
ages = [25, 30, 35]
people = [(name, age) for name, age in zip(names, ages)]
print(people)
```

This will output:

```
[('Alice', 25), ('Bob', 30), ('Charlie', 35)]
```

We can unpack tuples directly in the comprehension:

```
data = [(1, 'apple'), (2, 'banana'), (3, 'cherry')]
fruits = [fruit.upper() for _, fruit in data]
print(fruits)
```

This will output:

```
['APPLE', 'BANANA', 'CHERRY']
```

Remember the throw-away variable `_`? Here is a perfect example of how we would use it. The underscore (`_`) is used to ignore the first element of each tuple.



Sets

In this section, we're going to explore a useful data structure that might remind you of something from

math class: Sets. Don't worry, though – it's not as scary as it sounds!

A set in Python is a collection of unique elements. Think of it like a bag where you can put different items, but you can't put the same item in twice.

Creating Sets

Sets in Python are unordered collections of unique elements. This means two things - 1) the order of elements doesn't matter, 2) each element can appear only once in a set.

We can create a set using curly braces `{}`:

```
fruits = {"apple", "banana", "cherry"}
print(fruits)
```

This creates a set with three fruit names. Notice how we use curly braces, not square brackets like we do with lists.

Here's something tricky: you might think an empty set would be `{}`, but that actually creates an empty dictionary (next section)! To create an empty set, we use the `set()` constructor.

```
empty_set = set()
print(type(empty_set)) # This will print: <class 'set'>
```

We can also use `set()` to create sets with initial values from lists, tuples, or even strings.

```
# From a list
list_set = set([1, 2, 3, 4, 5])

# From a tuple
tuple_set = set((1, 2, 3, 4, 5))

# From a string
string_set = set("hello")

print(list_set)
print(tuple_set)
print(string_set) # Notice how each character becomes a
separate element
```

One of the cool things about sets is that they automatically remove duplicates.

```
numbers_with_duplicates = set([1, 2, 2, 3, 3, 3, 4, 5,
5])
print(numbers_with_duplicates) # This will print: {1,
2, 3, 4, 5}
```

This can be really useful when you want to get rid of duplicate elements in a list! Imagine you have a list of email addresses, and you want to remove any

duplicates. You could convert the list to a set and then back to a list:

```
emails = ["user@example.com", "another@example.com",
"user@example.com", "third@example.com"]
unique_emails = list(set(emails))
print(unique_emails)
```

Set Properties

Sets are a really useful data structure that can help you solve many programming problems efficiently. Think of a set like a bag where you can toss in various items, but with some special rules. Let's explore these rules and see how we can work with sets in Python.

Sets are unordered collections of unique elements. This means that unlike lists, sets don't maintain a specific order of elements.

```
my_set = {3, 1, 4, 1, 5, 9}
print(my_set)
```

When you run this code, you might see something like the list below or something completely different.

```
{1, 3, 4, 5, 9}
```

Notice how the order isn't the same as how we entered it? That's because sets don't care about order.

This has a few implications:

- ❑ You can't access elements by index (like you can with lists).
- ❑ When you iterate over a set, the order might be different each time.
- ❑ The order you see when you print a set might change between different runs of your program.

We already saw that sets only store unique elements. If you try to add a duplicate, it simply won't be added.

This automatic deduplication can be super helpful when you need to remove duplicates from a collection. However, there's a catch! Only hashable elements can be added to a set. In simple terms, this means immutable types like numbers, strings, and tuples can be elements of a set, but mutable types like lists can't.

```
valid_set = {1, "hello", (1, 2)} # This works
invalid_set = {[1, 2]} # This will raise an error
```

Sets themselves are mutable, which means you can add or remove elements after creating the set. However, there's also an immutable version called `frozenset`, but we won't go into more detail on that in this course.

```
# Mutable set
mutable_set = {1, 2, 3}
mutable_set.add(4)
print(mutable_set) # Output: {1, 2, 3, 4}

# Immutable frozenset
frozen = frozenset([1, 2, 3])
# frozen.add(4) # This would raise an error
```

Adding Elements to Sets

To add a single element to a set, use the `add()` method:

```
my_set = {1, 2, 3}
my_set.add(4)
print(my_set) # Output: {1, 2, 3, 4}

# Adding an existing element does nothing
my_set.add(2)
print(my_set) # Output: {1, 2, 3, 4}
```

To add multiple elements at once, use the `update()` method. This method can take any iterable (like a list, another set, or a tuple) and add all its elements to the set:

```
my_set = {1, 2, 3}
my_set.update([3, 4, 5])
print(my_set) # Output: {1, 2, 3, 4, 5}

# You can also update with multiple iterables
my_set.update({6, 7}, [8, 9])
print(my_set) # Output: {1, 2, 3, 4, 5, 6, 7, 8, 9}
```

Removing Elements from Sets

Sometimes, we need to remove elements from our sets. Python gives us several ways to do this.

The `remove()` method lets you take out a specific element from your set. Here's how it works:

```
my_fruits = {"apple", "banana", "cherry"}
my_fruits.remove("banana")
print(my_fruits) # Output: {'apple', 'cherry'}
```

But be careful! If you try to remove an element that's not in the set, Python will raise a `KeyError`:

```
my_fruits.remove("mango") # This will raise a KeyError
```

If you want a safer way to remove elements without worrying about errors, use the `discard()` method:

```
my_fruits = {"apple", "banana", "cherry"}
my_fruits.discard("banana")
print(my_fruits) # Output: {'apple', 'cherry'}

my_fruits.discard("mango") # This won't raise an error
print(my_fruits) # Output: {'apple', 'cherry'}
```

Use `discard()` when you're not sure if an element is in the set and you don't want your program to crash.

The `pop()` method removes and returns an arbitrary element from the set. The element removed by `pop()` is unpredictable. Use this method when you don't care which element is removed.

```
my_fruits = {"apple", "banana", "cherry"}
removed_fruit = my_fruits.pop()
print(f"Removed: {removed_fruit}")
print(f"Remaining fruits: {my_fruits}")
```

If you try to `pop()` from an empty set, you'll get a `KeyError`. Always check if the set is not empty before using `pop()`.

To remove all elements from a set, use the `clear()` method. This is useful when you want to reuse a set variable but start fresh with no elements.

Set Operations

Sets in Python support various mathematical operations – so, let's explore them!

Union (|)

The union combines elements from two or more sets, creating a new set with all unique elements.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
union_set = set1 | set2
print(union_set) # Output: {1, 2, 3, 4, 5}
```

This is great for merging collections without duplicates.

Intersection (&)

The intersection finds common elements between sets. Use this when you want to find items that exist in all sets.

```
set1 = {1, 2, 3, 4}
set2 = {3, 4, 5, 6}
common_elements = set1 & set2
print(common_elements) # Output: {3, 4}
```

Difference (-)

The difference operation gives you elements in one set but not in another.

```
set1 = {1, 2, 3, 4}
set2 = {3, 4, 5, 6}
difference = set1 - set2
print(difference) # Output: {1, 2}
```

Remember, order matters! `set1 - set2` is different from `set2 - set1`.

Symmetric Difference (^)

This operation returns elements that are in either set, but not in both:

```
set1 = {1, 2, 3, 4}
set2 = {3, 4, 5, 6}
sym_diff = set1 ^ set2
print(sym_diff) # Output: {1, 2, 5, 6}
```

It's like combining the unique elements from both sets.

This should give you a good idea of the operations that you can perform on sets. Remember to practice these concepts to get comfortable using them in your code. Sets are powerful tools in data science and these operations can help you manipulate data efficiently.

Set Methods for Operations

In addition to the set operations, there also exist equivalent set methods. These methods are super useful when you need to combine sets, find common elements, or identify differences between sets and work in the same way that operations work.

■ The `union()` method combines two or more sets, giving us a new set with all unique elements from all the sets. The operator equivalent is `|`.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
combined_set = set1.union(set2)
print(combined_set) # Output: {1, 2, 3, 4, 5}
```

Think of this like mixing different coloured marbles from separate bags into one big bag.

■ The `intersection()` method finds common elements between sets, just like the `&` operator.

```
set1 = {1, 2, 3, 4}
set2 = {3, 4, 5, 6}
common_elements = set1.intersection(set2)
print(common_elements) # Output: {3, 4}
```

■ The `difference()` method gives us elements that are in one set but not in the other. The operatorf equivalent is, you guessed it, `-`.

```
set1 = {1, 2, 3, 4}
set2 = {3, 4, 5, 6}
unique_to_set1 = set1.difference(set2)
print(unique_to_set1) # Output: {1, 2}
```

This is like finding out which toys you have that your friend doesn't.

■ Finally, the `symmetric_difference()` returns elements that are in either set, but not in both (^).

```
set1 = {1, 2, 3, 4}
set2 = {3, 4, 5, 6}
not_in_both = set1.symmetric_difference(set2)
print(not_in_both) # Output: {1, 2, 5, 6}
```

Now, here's something cool: these methods can work with multiple sets at once!

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
set3 = {5, 6, 7}
result = set1.union(set2, set3)
print(result) # Output: {1, 2, 3, 4, 5, 6, 7}
```

But wait, there's more! Python also gives us "in-place" versions of these methods. These modify the original set instead of creating a new one:

- `update()` (in-place union)
- `intersection_update()`
- `difference_update()`
- `symmetric_difference_update()`

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
set1.update(set2)
print(set1) # Output: {1, 2, 3, 4, 5}
```

Notice how `set1` changed? That's the "in-place" part in action! These in-place methods are great when you want to modify your original set and don't need to keep the old version.

Sets and these operations are super useful in real-world programming. You might use them to find common friends between users in a social network, compare inventory between stores, or filter unique data in data analysis.

Subset and Superset

In set theory, we often want to know if one set is contained within another or if it contains another set. This is where subsets and supersets come in handy. Python gives us easy ways to check these relationships.

Let's start with the `issubset()` and `issuperset()` methods. These are like asking, "Hey, is everything in this set also in that other set?" or "Does this set contain everything in that other set?"

```
fruits = {'apple', 'banana', 'cherry'}
citrus = {'lemon', 'orange'}
fruit_basket = {'apple', 'banana', 'cherry', 'lemon', 'orange'}
```

```
print(fruits.issubset(fruit_basket)) # True
print(citrus.issubset(fruit_basket)) # True
print(fruit_basket.issuperset(fruits)) # True
print(fruit_basket.issuperset(citrus)) # True
```

In this example, both `fruits` and `citrus` are subsets of `fruit_basket`, and `fruit_basket` is a superset of both `fruits` and `citrus`.

These methods return a boolean value - `True` if the relationship holds, `False` if it doesn't.

Python also gives us operators to check subset relationships: `<=` for subset and `<` for proper subset.

- `<=` checks if one set is a subset of another (can be equal)
- `<` checks if one set is a proper subset of another (must be smaller)

```
A = {1, 2, 3}
B = {1, 2, 3, 4, 5}
C = {1, 2, 3}

print(A <= B) # True (A is a subset of B)
print(A < B)  # True (A is a proper subset of B)
print(A <= C) # True (A is a subset of C)
print(A < C)  # False (A is not a proper subset of C,
              # they're equal)
```

Remember, a proper subset must be strictly smaller than the other set. If they're equal, it's still a subset, but not a proper subset.

Just like with subsets, we have operators for supersets: `>=` for superset and `>` for proper superset.

- `>=` checks if one set is a superset of another (can be equal)
- `>` checks if one set is a proper superset of another (must be larger)

```
X = {1, 2, 3, 4, 5}
Y = {1, 2, 3}
Z = {1, 2, 3, 4, 5}

print(X >= Y) # True (X is a superset of Y)
print(X > Y)  # True (X is a proper superset of Y)
print(X >= Z) # True (X is a superset of Z)
print(X > Z)  # False (X is not a proper superset of Z)
```

Imagine you're building a game where players collect items. You could use sets to represent item collections and check if a player has all the items needed for a special power-up:

```
required_items = {'sword', 'shield', 'potion'}
player_items = {'sword', 'shield', 'potion', 'map'}

if required_items.issubset(player_items):
    print("Player can use the special power-up!")
else:
    print("Player needs more items for the power-up.")
```

Checking Membership

In addition to looking for subsets and supersets, you'll often need to check if an item is present. Python makes this easy with the `'in'` operator.

```
fruits = {"apple", "banana", "cherry"}
print("banana" in fruits) # Output: True
print("grape" in fruits)  # Output: False
```

The `'in'` operator returns True if the item is in the set, and False if it's not. This is incredibly fast, especially for large sets.

```
large_set = set(range(1000000))
large_list = list(range(1000000))

# Using a set
print(999999 in large_set) # Very fast!

# Using a list
print(999999 in large_list) # Much slower
```

In this example, checking membership in the set is much faster than in the list. This is because sets use a technique called "hashing" to store their elements, which allows for very quick lookups. Remember though, that while sets are fast for membership testing, remember that they don't maintain any specific order of elements.

Sets in Boolean Context

We will now explore how sets behave in boolean contexts, how we can use randomness with sets, and how to perform set algebra operations. Let's dive in!

Sets can be used in boolean expressions, which means we can use them in situations where we need a True or False value. Here's how it works:

- ☑ An empty set is considered False
- ☑ Any non-empty set is considered True

This behaviour is really useful in conditional statements. Let's look at some examples:

```
# Empty set
empty_set = set()
if empty_set:
    print("This won't be printed")
else:
    print("The set is empty")

# Non-empty set
colours = {"red", "blue", "green"}
if colours:
    print("The set has some colours")
else:
    print("This won't be printed")
```

This feature is handy when you want to check if a set contains any elements before performing operations on it.

Disjoint Sets

In set theory, two sets are considered "disjoint" if they have no elements in common. Python provides a handy method to check for this: `isdisjoint()`.

Let's look at an example:

```
set1 = {1, 2, 3}
set2 = {4, 5, 6}
set3 = {3, 4, 5}

print(set1.isdisjoint(set2)) # True
print(set1.isdisjoint(set3)) # False
```

In this example, `set1` and `set2` are disjoint because they have no common elements. However, `set1` and `set3` are not disjoint because they both contain the element 3.

The `isdisjoint()` method is particularly useful in algorithm design. For instance, you might use it to check if two groups of items are completely separate.

Sets in List Comprehensions

Sets can be powerful tools when used in list comprehensions, which we saw earlier. They're especially useful for filtering out duplicate elements efficiently.

```
numbers = [1, 2, 2, 3, 3, 4, 5, 5]
unique_numbers = list({x for x in numbers})
print(unique_numbers) # [1, 2, 3, 4, 5]
```

In this code, we use a set comprehension `{x for x in numbers}` to create a set of unique numbers, then convert it back to a list.

We can also combine set operations with list generation:

```
set1 = {1, 2, 3, 4, 5}
set2 = {4, 5, 6, 7, 8}
common_elements = [x for x in set1 if x in set2]
print(common_elements) # [4, 5]
```

This creates a list of elements that are in both `set1` and `set2`.

Using sets in this way can be much faster than using lists, especially for large datasets. Sets use hash tables internally, which allows for very fast lookup times.

Best Practices for Using Sets

When should you choose sets over lists or other data structures? Here are some guidelines:

- Use sets when you need to store unique elements and don't care about order.
- Use sets when you frequently need to check if an item is in the collection.
- Use sets for mathematical set operations like union, intersection, and difference.

Set-based solutions are much faster, especially for large lists. But remember, while sets are powerful, they're not always the best choice. Lists are better when you need to maintain order or allow duplicate elements.

And that's it for sets. You'll hopefully appreciate how useful and important sets are in data science and handling data in Python. They can help you efficiently handle unique values, compare datasets, and analyse text characteristics. Practice using sets in your own projects to become more familiar with their capabilities.



Dictionaries

Last but not least, we will now explore a powerful and flexible data structure that you'll find incredibly useful in your programming journey: dictionaries.

A dictionary is a collection of items that stores data as key-value pairs. Think of it like a real dictionary, where you have a word (the key) and its definition (the value).

Now, let's break down the key-value pair concept:

- Keys are unique identifiers. Just like in a real dictionary where each word appears only once, in a Python dictionary, each key must be unique.
- Values are the data associated with each key. These can be of any data type - strings, numbers, lists, or even other dictionaries!
- We use a colon `:` to separate the key from its value, and commas to separate different key-value pairs.

Unlike lists, which use integer indices, dictionaries use keys to access values. This makes them more flexible for certain types of data. Dictionaries are therefore incredibly versatile. You might use them to store user information, configuration settings, or to count occurrences of items, among many other applications.

Creating Dictionaries

The simplest way to create a dictionary is by using curly braces `{}`. Inside these braces, we put our key-value pairs, separated by commas.

```
# Creating a dictionary of fruits and their colours
fruits = {"apple": "red", "banana": "yellow", "grape": "purple"}
print(fruits)
```

In this example, the fruit names are the keys, and their colours are the values. When you run this code, you'll see your dictionary printed out.

Another way to create a dictionary is by using the `dict()` constructor. This is a built-in Python function that creates dictionaries.

```
# Creating the same fruit dictionary using dict()
fruits = dict(apple="red", banana="yellow", grape="purple")
print(fruits)
```

This method is handy when your keys are simple strings. The result is exactly the same as our first example.

Sometimes, you might have your key-value pairs in a list of tuples or another sequence. You can turn these into a dictionary too using the `dict()` method.

```
# Creating a dictionary from a list of tuples
fruit_list = [("apple", "red"), ("banana", "yellow"), ("grape", "purple")]
fruits = dict(fruit_list)
print(fruits)
```

This is super useful when you're working with data that's already in a specific format. Imagine you're making a simple game where characters have different attributes. You could use a dictionary to store this information:

```
character = {
    "name": "Hero",
    "health": 100,
    "weapon": "sword",
    "armor": "shield"
}
print(f"The {character['name']} has {character['health']} health points.")
```

There are some rules to follow when creating dictionaries though:

- Dictionary keys must be unique. If you try to use the same key twice, the last value will overwrite any previous ones.
- Keys in dictionaries are case-sensitive. So "Apple" and "apple" would be considered different keys.
- While we've used strings as keys in these examples, remember that dictionary keys can be any immutable type in Python, like numbers or tuples.

That's it for creating dictionaries – super easy!

Accessing Dictionary Elements

The most straightforward way to access a value in a dictionary is by using its key. Here's how it works:

```
# Let's create a simple dictionary
student = {"name": "Alice", "age": 20, "grade": "A"}

# To access a value, we use square brackets with the key
print(student["name"]) # This will print: Alice
print(student["age"])  # This will print: 20
```

This method is quick and easy, but it has a drawback. If you try to access a key that doesn't exist, Python will raise a `KeyError`. For example:

```
print(student["height"]) # This will raise a KeyError
```

To avoid potential `KeyErrors`, Python provides a handy method called `get()`. Here's how to use it:

```
# Using the same dictionary as before
print(student.get("name")) # This will print: Alice
print(student.get("height")) # This will print: None
```

The big advantage of `get()` is that it doesn't raise an error if the key doesn't exist. Instead, it returns `None`.

You can also specify a default value to return if the key isn't found. This is super useful when you're not sure

if a key exists and you want to provide a fallback value.

```
print(student.get("height", "Not available"))
# This will print: Not available
```

Modifying Dictionaries

To add a new item to a dictionary, we simply assign a value to a new key. Here's how:

```
# Let's start with a dictionary of fruits and their
# colours
fruits = {"apple": "red", "banana": "yellow"}
print(fruits) # Output: {"apple": "red", "banana":
# "yellow"}

# Now, let's add a new fruit
fruits["orange"] = "orange"
print(fruits) # Output: {"apple": "red", "banana":
# "yellow", "orange": "orange"}
```

See how easy that was? We just used square brackets with the new key and assigned it a value if we want to change the value of an existing key, is just as simple.

```
fruits["apple"] = "green" # Let's change the colour of
our apple
print(fruits) # Output: {"apple": "green", "banana":
# "yellow", "orange": "orange"}
```

We used the same syntax as adding, but since "apple" already existed, its value was updated.

Python also provides a handy method called `update()` that allows us to add multiple key-value pairs at once or update existing ones:

```
fruits.update({"grape": "purple", "apple": "red"})
print(fruits)
# Output: {"apple": "red", "banana": "yellow", "orange":
# "orange", "grape": "purple"}
```

Notice how it added "grape" and updated "apple" back to "red" in one go!

Removing Items from Dictionaries

Now that we know how to add and modify items, let's learn how to remove them. Python gives us several ways to do this:

- The `pop()` method removes an item with a specified key and returns its value:

```
removed_fruit = fruits.pop("banana")
print(removed_fruit) # Output: "yellow"
print(fruits) # Output: {"apple": "red", "orange":
# "orange", "grape": "purple"}
```

If you try to pop a key that doesn't exist, you'll get an error. To avoid this, you can provide a default value:

```
safe_remove = fruits.pop("mango", "Not found")
print(safe_remove) # Output: "Not found"
```

■ The `popitem()` method, on the other hand, removes and returns the last inserted key-value pair as a tuple:

```
last_item = fruits.popitem()
print(last_item) # Output: ("grape", "purple")
print(fruits) # Output: {"apple": "red", "orange":
               # "orange"} because fruits no longer
               # exists
```

Practical use case: Imagine you're building a simple inventory system for a small store. You could use a dictionary to keep track of items and their quantities.

```
inventory = {"apples": 50, "bananas": 30, "oranges": 40}

# A customer buys 5 apples
inventory["apples"] -= 5

# You receive a shipment of 20 more bananas
inventory["bananas"] += 20

# You decide to stop selling oranges
inventory.pop("oranges")

# You get a new product
inventory["grapes"] = 25

print(inventory)
# Output: {"apples": 45, "bananas": 50, "grapes": 25}
```

Dictionary Methods

Python gives us some handy methods to work with dictionaries. Let's check them out!

- `keys()`: Returns all keys in the dictionary
- `values()`: Returns all values in the dictionary
- `items()`: Returns all key-value pairs as tuples

```
student = {'name': 'Alice', 'age': 20, 'grade': 'A'}

print(student.keys()) # Output: dict_keys(['name',
                        # 'age', 'grade'])
print(student.values()) # Output: dict_values(['Alice',
                                                # 20, 'A'])
print(student.items()) # Output: dict_items([('name',
                                                # 'Alice'), ('age', 20),
                                                # ('grade', 'A')])
```

These methods are great for looping over dictionaries:

```
for key in student.keys():
    print(key)

for value in student.values():
    print(value)
```

```
for key, value in student.items():
    print(f"{key}: {value}")
```

When you want to create a copy of a dictionary, use the `copy()` method:

```
original = {'a': 1, 'b': 2}
copied = original.copy()

copied['c'] = 3
print(original) # Output: {'a': 1, 'b': 2}
print(copied) # Output: {'a': 1, 'b': 2, 'c': 3}
```

This creates a shallow copy. If your dictionary contains other objects (like lists), those objects are not copied, just referenced.

The `fromkeys()` method is useful when you want to create a new dictionary with known keys and a default value. Here we use a list of keys: 'a', 'b', 'c' and a default value of 0 to create a new dictionary using the `fromkeys()` method.

```
keys = ['a', 'b', 'c']
default_value = 0

new_dict = dict.fromkeys(keys, default_value)
print(new_dict) # Output: {'a': 0, 'b': 0, 'c': 0}
```

Dictionary Comprehensions

Dictionary comprehensions are a concise way to create dictionaries. They're similar to list comprehensions but for dictionaries.

The basic syntax is: `{key_expression: value_expression for item in iterable}`

```
squares = {x: x**2 for x in range(5)}
print(squares)
# Output: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

You can also add a conditional `if`:

```
even_squares = {x: x**2 for x in range(10) if x % 2 == 0}
print(even_squares)
# Output: {0: 0, 2: 4, 4: 16, 6: 36, 8: 64}
```

You can use dictionary comprehensions to modify existing dictionaries:

```
prices = {'apple': 0.5, 'banana': 0.25, 'orange': 0.75}
doubled_prices = {fruit: price * 2 for fruit, price in
                  prices.items()}
print(doubled_prices)
# Output: {'apple': 1.0, 'banana': 0.5, 'orange': 1.5}
```

Dictionary comprehensions can make your code more concise, but don't sacrifice readability for brevity. If a comprehension becomes too complex, it's often better to use a traditional for loop.

When using `fromkeys()`, remember that all keys will reference the same default value object. This can lead to unexpected behaviour with mutable objects like lists.

Dictionary and Set View Objects

When you use `keys()`, `values()`, or `items()`, Python doesn't actually create a new list of these elements. Instead, it creates something called a "view object". This is a dynamic view of the dictionary's entries, which means if the dictionary changes, the view reflects these changes.

```
my_dict = {"a": 1, "b": 2}
keys_view = my_dict.keys()
print(keys_view) # dict_keys(['a', 'b'])

my_dict["c"] = 3
print(keys_view) # dict_keys(['a', 'b', 'c'])
```

As you can see, when we added a new key-value pair to `my_dict`, the `keys_view` automatically updated to include the new key.

This is really useful because it saves memory and keeps everything in sync. However, if you need a static list of the keys, values, or items, you can convert the view to a list.

Merging Dictionaries

Sometimes, you might want to combine two dictionaries. There are a couple of ways to do this:

■ The `update()` method adds the key-value pairs from one dictionary to another:

```
dict1 = {"a": 1, "b": 2}
dict2 = {"c": 3, "d": 4}
dict1.update(dict2)
print(dict1) # {'a': 1, 'b': 2, 'c': 3, 'd': 4}
```

If there are any duplicate keys, the values from the dictionary being added (`dict2` in this case) will overwrite the existing values.

■ You can use the `|` operator to merge dictionaries just as with sets:

```
dict1 = {"a": 1, "b": 2}
dict2 = {"c": 3, "d": 4}
merged_dict = dict1 | dict2
print(merged_dict) # {'a': 1, 'b': 2, 'c': 3, 'd': 4}
```

This creates a new dictionary without modifying the original ones. Imagine you have two dictionaries of user preferences, maybe one from a local database and

one from a cloud service. You could use these methods to combine them into a single, updated dictionary.

Dictionaries vs. Lists of Tuples

Let's explore some more advanced concepts related to dictionaries, comparing them with other data structures, and learning how to use them in different scenarios.

Both dictionaries and lists of tuples can be used to store key-value pairs, but they have different strengths.

- Use dictionaries when you need fast lookups based on unique keys.
- Use lists of tuples when order matters or when you might have duplicate keys.

Here's an example to illustrate:

```
# Dictionary
student_scores = {
    "Alice": 95,
    "Bob": 87,
    "Charlie": 92
}

# List of tuples
student_scores_list = [
    ("Alice", 95),
    ("Bob", 87),
    ("Charlie", 92)
]
```

With the dictionary, you can quickly access a score:

```
print(student_scores["Bob"]) # Output: 87
```

With the list of tuples, you'd need to loop through to find a score:

```
for name, score in student_scores_list:
    if name == "Bob":
        print(score)
        break
```

Sorting Dictionaries

Dictionaries in Python are unordered by default, but sometimes we need to work with their contents in a specific order. To sort a dictionary by its keys, we can use the `sorted()` function.

```
my_dict = {'b': 2, 'c': 3, 'a': 1}
sorted_keys = sorted(my_dict.keys())
print(sorted_keys) # Output: ['a', 'b', 'c']

# To get a new dictionary with sorted keys:
sorted_dict = {k: my_dict[k] for k in sorted_keys}
print(sorted_dict) # Output: {'a': 1, 'b': 2, 'c': 3}
```

Inverting a Dictionary

Inverting a dictionary means swapping its keys and values. Here's how we can do it:

```
original_dict = {'a': 1, 'b': 2, 'c': 3}
inverted_dict = {v: k for k, v in original_dict.items()}
print(inverted_dict) # Output: {1: 'a', 2: 'b', 3: 'c'}
```

This method works well when values are unique. If there are duplicate values in the original dictionary, some data may be lost in the inversion process.

Dictionary and JSON

JSON (JavaScript Object Notation) is a popular data format used for storing and exchanging data – we'll cover this format in detail in a later chapter. Python dictionaries are closely related to JSON objects. We can use the `json` module to convert a Python dictionary to a JSON string with the `json.dumps()` function.

```
import json

my_dict = {'name': 'Alice', 'age': 30, 'city': 'New York'}
json_string = json.dumps(my_dict)
print(json_string)
# Output: {"name": "Alice", "age": 30, "city": "New York"}
```

We can also convert a JSON string back to a Python dictionary using the `json.loads()` method.

```
json_string = '{"name": "Bob", "age": 25, "city": "London"}'
my_dict = json.loads(json_string)
print(my_dict)
# Output: {'name': 'Bob', 'age': 25, 'city': 'London'}
```

This ability to convert between dictionaries and JSON makes it easy to work with data from web services or configuration files.

Handling Missing Keys

We have now seen that dictionaries are incredibly versatile data structures in Python. They allow us to store and retrieve data using key-value pairs. Let's explore some more advanced techniques and practical uses for dictionaries.

Sometimes, we need to work with keys that might not exist in our dictionary. Python provides a couple of handy methods to deal with this situation safely.

■ The `setdefault()` method allows us to check for a key and set a default value if it doesn't exist, all in one step.

```
user_info = {'name': 'Alice', 'age': 30}

# Get the 'city' if it exists, otherwise set it to 'Unknown'
city = user_info.setdefault('city', 'Unknown')
print(city) # Output: Unknown
print(user_info)
# Output: {'name': 'Alice', 'age': 30, 'city': 'Unknown'}
```

■ The `get()` method is another way to safely retrieve a value, providing a default if the key doesn't exist.

```
user_info = {'name': 'Bob', 'age': 25}

# Get the 'city' if it exists, otherwise return 'Not specified'
city = user_info.get('city', 'Not specified')
print(city) # Output: Not specified
print(user_info) # Output: {'name': 'Bob', 'age': 25}
```

The key difference between `setdefault()` and `get()` is that `setdefault()` modifies the dictionary if the key doesn't exist, while `get()` doesn't change the original dictionary.

Real-world Scenarios

Dictionaries are used extensively in real-world programming. Imagine you're analysing sales data:

```
sales_data = [
    {'date': '2023-01-01', 'product': 'laptop',
     'quantity': 5, 'price': 1000},
    {'date': '2023-01-01', 'product': 'mouse',
     'quantity': 10, 'price': 25},
    {'date': '2023-01-02', 'product': 'laptop',
     'quantity': 3, 'price': 1000},
]

product_sales = {}
for sale in sales_data:
    product = sale['product']
    total = sale['quantity'] * sale['price']
    product_sales[product] = product_sales.get(product, 0) + total
print(product_sales) # Output: {'laptop': 8000, 'mouse': 250}
```



Functions

Give yourself a big pat on the back. You've already covered a big part of Python syntax, with variables, data structures (lists, tuples, dictionaries), and control structures (loops and conditions). Well done!

You are now ready to take things a step further by looking at functions! While you doing the exercises and practical projects, have you ever found yourself writing the same code over and over again? This is where functions would have come in handy.

A function is like a mini-program within your main program. It's a reusable block of code that performs a specific task. Think of a function as a black box: you give it some input, it does some work, and then it gives you an output.

Why do we use functions?

- Code Reusability: Write once, use many times!
- Modularity: Break your program into smaller, manageable pieces.
- Abstraction: Hide complex operations behind a simple interface.

These benefits lead us to an important principle in programming: DRY (Don't Repeat Yourself). This is a very important principle in computational science, regardless of the language.

DRY is all about avoiding code duplication. Here's why it's important:

- Easier Maintenance: You only have to change the code in one place and it affects all uses.

- Consistency: It ensures the same logic is used everywhere.
- Time-Saving: You only write code once, and use it multiple times.

Defining Functions

A function is a block of code that performs a specific task. It's a way to package up a set of instructions that you can use over and over again. Let's look at how to create a function in Python.

To define a function, we use the `def` keyword. Here's the basic structure:

```
def function_name(parameters):
    # Function body
    # Your code goes here
    return result # Optional
```

Let's break this down:

- `def`: This keyword tells Python that we're defining a function.
- `function_name`: This is the name you give to your function. We'll talk more about naming in a moment.
- `parameters`: These are inputs that the function can accept (optional).
- `colon (:)`: This marks the end of the function header.
- Function body: This is where you write the code that the function will execute. It must be indented.
- `return`: This keyword is used to specify what the function should output (optional).

Here's a simple example:

```
def greet(name):
    message = f"Hello, {name}!"
    return message

# Using the function
result = greet("Alice")
print(result) # Output: Hello, Alice!
```

In this example, `greet` is the function name, `name` is the parameter, and the function returns a greeting message.

Choosing good function names is important for making your code readable. Here are some guidelines:

- Use lowercase letters and underscores: `calculate_average`, not `CalculateAverage`.
- Be descriptive: `calculate_total_price`, not `calc_price`.
- Use verb-based names: `get_user_input`, `process_data`.
- Avoid reserved keywords: Don't use names like `if`, `for`, `while`, etc.

Here's an example of good vs. not-so-good function names:

```
# Good function names
def calculate_area(length, width):
    return length * width

def get_user_name():
    return input("Enter your name: ")

# Not-so-good function names
def Area(l, w):
    # Capitalization and unclear parameter names
    return l * w

def do_stuff(): # Not descriptive
    # some code here
```

Function Parameters and Arguments

Let's explore different ways to pass information into functions, making them more flexible and powerful.

- Positional arguments are the most basic way to pass data into a function. They're called "positional" because the order in which you provide them matters.

```
def greet(name, age):
    print(f"Hello, {name}! You are {age} years old.")

greet("Alice", 25)
```

In this function, `name` and `age` are parameters, and when we call the function, `"Alice"` and `25` are the arguments. The order is important - `"Alice"` matches with `name`, and `25` matches with `age`.

What if we want to pass a variable number of arguments? That's where `*args` comes in handy.

```
def print_args(*args):
    for arg in args:
        print(arg)

print_args("apple", "banana", "cherry")
```

This function can accept any number of positional arguments. The `*args` parameter collects them into a tuple.

- Keyword arguments are passed to a function using name-value pairs. They offer more flexibility and can make your code more readable.

```
def describe_pet(animal, name):
    print(f"I have a {animal} named {name}.")

describe_pet(animal="dog", name="Rex")
describe_pet(name="Whiskers", animal="cat")
```

See how we can change the order when using keyword arguments? This can be especially helpful in functions with many parameters.

For functions that might receive an unknown number of keyword arguments, we use `**kwargs`:

```
def print_kwargs(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

print_kwargs(name="Alice", age=25, city="New York")
```

This function can accept any number of keyword arguments and prints them as key-value pairs.

- Default values allow a function to work even if you don't provide all the arguments. Here's how it looks:

```
def greet(name, greeting="Hello"):
    print(f"{greeting}, {name}!")

greet("Bob")
greet("Alice", "Good morning")
```

In this example, if we don't provide a greeting, it defaults to `"Hello"`. However, be careful with mutable default values like lists or dictionaries. They can lead to unexpected behaviour. For example:

```
def add_item(item, list=[]):
    list.append(item)
    return list

print(add_item("apple"))
print(add_item("banana"))
```

You might expect the second call to return just `["banana"]`, but it actually returns `["apple", "banana"]`. This is because the default list is created only once when the function is defined.

To avoid this, use `None` as the default and create the list inside the function:

```
def add_item(item, list=None):
    if list is None:
        list = []
    list.append(item)
    return list

print(add_item("apple"))
print(add_item("banana"))
```

Now each call to the function will start with a fresh, empty list.

Remember, you can always override default values by providing your own arguments when calling the function.

Return Statements

Return statements are crucial in functions. They allow your function to send data back to wherever it was called from.

When you want your function to output a value, you use the `return` keyword. Here's a simple example:

```
def add_numbers(a, b):
    return a + b

result = add_numbers(5, 3)
print(result) # Output: 8
```

In this case, the function `'add_numbers'` returns the sum of `'a'` and `'b'`.

You can return a single value, as we did above, or multiple values. When returning multiple values, Python automatically packs them into a tuple:

```
def get_user_info():
    return "Alice", 25, "New York"

name, age, city = get_user_info()
print(f"{name} is {age} years old and lives in {city}")
# Output: Alice is 25 years old and lives in New York
```

You can use return statements to exit a function early if a certain condition is met:

```
def divide(a, b):
    if b == 0:
        return "Error: Cannot divide by zero"
    return a / b

print(divide(10, 2)) # Output: 5.0
print(divide(10, 0)) # Output: Error: Cannot divide by zero
```

Once a `return` statement is executed, the function immediately exits. No code after the return statement in that function will be executed.

If a function doesn't have a return statement, or if it has a return statement without a value, it implicitly returns `None`:

```
def greet(name):
    print(f"Hello, {name}!")

result = greet("Bob")
print(result) # Output: None
```

Docstrings

Docstrings are a great way to document your functions. They make your code more readable and easier to understand, both for yourself and for other programmers.

Docstrings are written using triple quotes (`'''` or `"""`) right after the function definition.

```
def calculate_area(length, width):
    """
    Calculate the area of a rectangle.

    Parameters:
    length (float): The length of the rectangle
    width (float): The width of the rectangle

    Returns:
    float: The area of the rectangle
    """
    return length * width
```

You can access a function's docstring using the `.doc` attribute or the `help()` function that we saw at the very beginning of the course:

```
print(calculate_area.__doc__)
# or
help(calculate_area)
```

When you write your docstring:

- ✓ Always include a brief description of what the function does.
- ✓ List and describe all parameters.
- ✓ Specify the return value(s).
- ✓ If applicable, include an example of how to use the function.

Here's an example putting it all together:

```
def get_grade(score):
    """
    Determine the letter grade based on a numerical score.

    Parameters:
    score (int): The numerical score, should be between 0 and 100

    Returns:
    str: The corresponding letter grade

    Example:
    >>> get_grade(85)
    'B'
    """
    if not 0 <= score <= 100:
        return "Invalid score"
    elif score >= 90:
        return "A"
    elif score >= 80:
        return "B"
    elif score >= 70:
        return "C"
    elif score >= 60:
```

```

        return "D"
    else:
        return "F"

# Using the function
print(get_grade(85)) # Output: B

# Accessing the docstring
print(get_grade.__doc__)

```

Remember, good documentation makes your code more professional and easier for others (including your future self) to understand and use.

Scope and Namespace

In Python, scope refers to the area of your code where a variable is visible and can be accessed. Understanding scope is crucial for writing clean and bug-free code. We have two main types of scope: local and global.

- 🟢 Local scope: Variables defined inside a function are in the local scope. They can only be accessed within that function.
- 🟢 Global scope: Variables defined outside of any function are in the global scope. They can be accessed from anywhere in your code.

Let's look at an example:

```

x = 10 # Global variable

def print_x():
    x = 20 # Local variable
    print("Local x:", x)

print_x()
print("Global x:", x)

```

Output:

```

Local x: 20
Global x: 10

```

In this example, we have two variables named 'x'. The one outside the function is global, while the one inside the function is local. Python uses the LEGB rule (Local, Enclosing, Global, Built-in) to determine which variable to use.

When you create a variable with the same name inside a function, it "shadows" the global variable. This means the local variable takes precedence within the function.

Sometimes, you might want to modify a global variable from within a function. To do this, you use the `global` keyword:

```
count = 0
```

```

def increment_counter():
    global count
    count += 1
    print("Counter:", count)

increment_counter()
increment_counter()

```

Output:

```

Counter: 1
Counter: 2

```

It's best to use global variables sparingly. They can make your code harder to understand and maintain as they can lead to unexpected behaviour and make debugging difficult.

For nested functions, Python provides the `nonlocal` keyword. It's used to work with variables in the nearest enclosing scope that isn't global.

```

def outer():
    x = "outer"

    def inner():
        nonlocal x
        x = "inner"
        print("Inner:", x)

    inner()
    print("Outer:", x)

outer()

```

Output:

```

Inner: inner
Outer: inner

```

The `nonlocal` keyword allows the inner function to modify the variable 'x' from the outer function's scope.

Functions as Objects

In Python, functions are what's called first-class objects. This means you can treat them like any other object, such as assigning them to variables or passing them as arguments to other functions.

You can assign a function to a variable, creating a new reference to the function:

```

def greet(name):
    return f"Hello, {name}!"

# Assign the function to a variable
say_hello = greet

# Use the new variable to call the function
print(say_hello("Alice"))
Hello, Alice!

```

This feature allows for dynamic dispatch, where you can choose which function to call based on certain conditions.

You can also pass functions as arguments to other functions. This is commonly used for callbacks and creating higher-order functions:

```
def apply_operation(x, y, operation):
    return operation(x, y)

def add(a, b):
    return a + b

def multiply(a, b):
    return a * b

print(apply_operation(5, 3, add))
print(apply_operation(5, 3, multiply))
```

Output:

```
8
15
```

This technique is powerful for creating flexible and reusable code. It's widely used in functional programming and for implementing strategies or plugins. You might use this when working with data processing pipelines, where different operations need to be applied to data based on certain conditions.

Lambda Functions

Lambda functions, also known as anonymous functions, are small, one-time-use functions that you can create on the fly. They're called "anonymous" because they don't have a name like regular functions do.

Syntax: The basic syntax for a lambda function is:

```
lambda arguments: expression
```

Here's a simple example:

```
add = lambda x, y: x + y
result = add(5, 3)
print(result) # Output: 8
```

In this example, we created a lambda function that adds two numbers and assigned it to the variable `add`. We then used it just like a regular function.

Lambda functions are great for short, simple operations, especially when you need a quick function for things like sorting or filtering. They're commonly used in functional programming concepts like `map()`, `filter()`, and `reduce()`.

While lambda functions are convenient, they have some limitations:

- ❑ They can only contain a single expression.
- ❑ They can be harder to read, especially for complex operations.
- ❑ Debugging can be challenging since they don't have a name.

So, use lambda functions sparingly. If your operation is complex or used multiple times, it's better to define a regular function.

Recursive Functions

Recursive functions are functions that call themselves within their own code. They're particularly useful for solving problems that have a recursive nature, like traversing tree structures or calculating factorials.

A recursive function typically has two parts:

- ❑ Base case: The condition where the function stops calling itself.
- ❑ Recursive case: Where the function calls itself with a modified input.

Here's a simple example calculating factorial:

```
def factorial(n):
    if n == 0 or n == 1: # Base case
        return 1
    else: # Recursive case
        return n * factorial(n - 1)

print(factorial(5)) # Output: 120
```

How it works:

- ❑ If `n` is 0 or 1, we return 1 (base case).
- ❑ Otherwise, we multiply `n` by the factorial of `(n-1)` (recursive case).
- ❑ This continues until we reach the base case.

Recursive functions can make code more readable for naturally recursive problems. They're great for working with hierarchical data structures like trees.

However, if the recursion goes too deep, you might run out of memory (stack overflow error). Recursive functions can also be slower than iterative solutions for some problems.

Lambda functions and recursive functions are powerful tools in Python. Lambda functions offer a concise way to write small, one-time-use functions,

while recursive functions provide an elegant solution for problems with a naturally recursive structure.

Higher-Order Functions

Higher-order functions are functions that can work with other functions. They can either take functions as arguments or return functions as results. This is a really powerful concept that allows for more flexible and reusable code. Let's look at some built-in Python functions that take functions as arguments.

The `map()` function applies a given function to each item in an iterable (like a list) and returns a map object.

```
def square(x):
    return x ** 2

numbers = [1, 2, 3, 4, 5]
squared_numbers = list(map(square, numbers))
print(squared_numbers) # Output: [1, 4, 9, 16, 25]
```

The `filter()` function creates a new list with elements that satisfy a given condition.

```
def is_even(x):
    return x % 2 == 0

numbers = [1, 2, 3, 4, 5, 6]
even_numbers = list(filter(is_even, numbers))
print(even_numbers) # Output: [2, 4, 6]
```

The `sorted()` function can take a key function to determine how to sort the elements.

```
words = ['apple', 'banana', 'cherry', 'date']
sorted_words = sorted(words, key=len)
print(sorted_words) # Output: ['date', 'apple', 'banana', 'cherry']
```

But, you can create your own higher-order functions too!

```
def apply_twice(func, value):
    return func(func(value))

def add_five(x):
    return x + 5

result = apply_twice(add_five, 10)
print(result) # Output: 20
```

Decorators

Decorators are a fantastic way to modify or enhance functions without changing their core logic. Think of them as wrappers that add extra functionality to your functions.

Basic Decorator Syntax and Usage

Decorators use the "@" symbol and are placed right above the function definition. Here's a simple example:

```
def my_decorator(func):
    def wrapper():
        print("Something is happening before the
function is called.")
        func()
        print("Something is happening after the function
is called.")
    return wrapper

@my_decorator
def say_hello():
    print("Hello!")

say_hello()
```

When you run this code, you'll see:

```
Something is happening before the function is called.
Hello!
Something is happening after the function is called.
```

The `@my_decorator` line is equivalent to `say_hello = my_decorator(say_hello)`. It wraps our function with additional behaviour.

Creating Decorators with Arguments

Sometimes, you want your decorators to accept arguments. Here's how you can do that:

```
def repeat(times):
    def decorator(func):
        def wrapper(*args, **kwargs):
            for _ in range(times):
                result = func(*args, **kwargs)
            return result
        return wrapper
    return decorator

@repeat(3)
def greet(name):
    print(f"Hello, {name}!")

greet("Alice")
```

This will print "Hello, Alice!" three times.

Multiple Decorators

You can stack multiple decorators on a single function. They're applied from bottom to top:

```
def bold(func):
    def wrapper():
        return "<b>" + func() + "</b>"
    return wrapper

def italic(func):
    def wrapper():
        return "<i>" + func() + "</i>"
    return wrapper
```

```

    return wrapper

@bold
@italic
def hello():
    return "Hello, World!"

print(hello()) # Outputs: <b><i>Hello, World!</i></b>

```

Function Annotations (Optional)

Function annotations allow you to add metadata to your function parameters and return values. While they don't affect the runtime behavior of your code, they can be useful for documentation and static type checking.

Adding Type Hints

Here's how you can add type hints to a function:

```

def greet(name: str) -> str:
    return f"Hello, {name}!"

print(greet("Bob")) # Outputs: Hello, Bob!

```

In this example, `: str` after name indicates that name should be a string, and `-> str` indicates that the function returns a string.

Optional vs Required Parameters

For optional parameters, you can use the `Optional` type from the `typing` module:

```

from typing import Optional

def greet(name: str, title: Optional[str] = None) -> str:
    if title:
        return f"Hello, {title} {name}!"
    return f"Hello, {name}!"

print(greet("Alice", "Ms.)) # Outputs: Hello, Ms. Alice!
print(greet("Bob")) # Outputs: Hello, Bob!

```

Remember, while these features are powerful, they should be used carefully. Clear, simple code is often better than clever, complex code.

Generator Functions

Generator functions allow you to generate a sequence of values over time, instead of computing them all at once and storing them in memory. We do this using the `yield` keyword.

```

def countdown(n):
    while n > 0:
        yield n
        n -= 1

```

```

for num in countdown(5):
    print(num)

# Output:
# 5
# 4
# 3
# 2
# 1

```

The `yield` keyword is what makes this a generator function. It returns a value and pauses the function's execution until the next value is requested.

Generators are great. They produce values on-the-fly, which is great for large datasets. Values are computed only when needed, which can improve performance. Finally, you can create generators for infinite sequences without running out of memory.

```

def infinite_counter():
    num = 0
    while True:
        yield num
        num += 1

counter = infinite_counter()
print(next(counter)) # 0
print(next(counter)) # 1
print(next(counter)) # 2

```

The `yield from` statement allows you to yield values from another generator, making it easier to compose generators.

```

def gen1():
    yield from range(3)
    yield from range(4, 6)

for num in gen1():
    print(num)

```

This is particularly useful for flattening nested generators or delegating to sub-generators. Imagine you're processing a large file. Instead of reading the entire file into memory, you could use a generator to read and process it line by line:

```

def read_large_file(file_path):
    with open(file_path, 'r') as file:
        for line in file:
            yield line.strip()

for line in read_large_file('very_large_file.txt'):
    # Process each line
    print(line)

```

This approach allows you to work with files that are larger than your computer's memory.

Callback Functions

Callback functions are functions that are passed as arguments to other functions and are executed after

some specific event or condition occurs. They are commonly used in event-driven programming and asynchronous operations. Here's a simple example:

```
def greet(name):
    print(f"Hello, {name}!")

def process_name(callback):
    name = input("Enter your name: ")
    callback(name)

process_name(greet)
```

In this example, `greet` is a callback function that's passed to `process_name`. After getting the user's name, `process_name` calls the callback function.

Callbacks are useful for handling asynchronous operations, like reading files or making API requests:

```
import time

def long_running_task(callback):
    print("Starting long task...")
    time.sleep(3) # Simulating a long operation
    callback("Task completed!")

def on_complete(result):
    print(result)

long_running_task(on_complete)
```

While callbacks are powerful, they can lead to complex code structures if overused. In modern Python, asynchronous functions often provide a more readable alternative for handling asynchronous operations.



Modules and Packages

A module in Python is simply a file that contains Python code. Think of it as a container for related functions, variables, and even classes. The key thing to remember is that a module is just a .py file - nothing more complicated than that!

You might be wondering, "Why bother with modules? Can't I just write all my code in one file?" Well, you could, but as your programs grow larger, that can get messy quickly. Modules help us in two main ways:

- **Code Organization:** Modules allow you to group related code together. For example, you might have a module for math functions, another for file operations, and so on.
- **Code Reusability:** Once you've written a module, you can use it in multiple programs without having to copy and paste the code.

Let's see a simple example. Imagine we have a file called `math_operations.py`:

```
def add(a, b):
    return a + b

def subtract(a, b):
    return a - b

def multiply(a, b):
    return a * b

def divide(a, b):
    if b != 0:
        return a / b
    else:
```

```
return "Error: Division by zero"
```

This file is now a module! Woohoo! We can use these functions in another file without having to rewrite them.

As your projects grow, modules become increasingly valuable. They allow you to break down complex problems into smaller, manageable parts. You can work on different parts of a program independently. Working with a module structure makes collaborations with other programmers much easier. It helps you to keep your main program file clean and easy to read.

For example, in a large game project, you might have separate modules for: Character creation, Game physics, Sound effects, User interface, etc... This way, each part of the game can be developed and tested independently.

Going back to the first example, imagine you're building a calculator application. You could put all your math functions in a module called `calculator_functions.py`. Then, in your main program, you can import and use these functions:

```
import calculator_functions

result = calculator_functions.add(5, 3)
print(result) # Output: 8

result = calculator_functions.multiply(4, 7)
print(result) # Output: 28
```

This keeps your main program clean and allows you to easily reuse your math functions in other projects.

Importing Modules

The simplest way to import a module is using the `import` keyword followed by the module name.

```
import random
```

This imports the entire `random` module. After importing, you can use functions from this module by writing the module name, followed by a dot, and then the function name. For example:

```
import random

random_number = random.randint(1, 10)
print(random_number)
```

This will print a random integer between 1 and 10.

Sometimes, you only need one or two functions from a module. In this case, you can import just those specific items:

```
from math import sqrt, pi

radius = 5
area = pi * (sqrt(radius) ** 2)
print(area)
```

Here, we're only importing the `sqrt` function and `pi` constant from the `math` module. We can then use these directly without referencing the module name.

If a module name is long or conflicts with something else in your code, you can give it a shorter or different name (an alias) when you import it:

```
import matplotlib.pyplot as plt

plt.plot([1, 2, 3, 4])
plt.show()
```

This is especially useful for modules with long names like `matplotlib.pyplot`, which we have an entire lesson on.

You can also import all items from a module using the asterisk `*`:

```
from math import *
```

This allows you to use all functions and variables from the `math` module without the module name prefix. However, this method is generally discouraged because:

- ❑ It can lead to naming conflicts if two modules have functions with the same name.
- ❑ It makes your code less readable because it's not clear where functions are coming from.
- ❑ It can accidentally overwrite variables in your code.

Let's say you're building a simple game that needs random numbers and some mathematical operations. You might start your script like this:

```
import random
from math import sqrt, pi

# Generate a random number between 1 and 10
player_strength = random.randint(1, 10)

# Calculate the area of a circular game board
board_radius = 5
board_area = pi * (sqrt(board_radius) ** 2)

print(f"Player strength: {player_strength}")
print(f"Game board area: {board_area:.2f}")
```

This script demonstrates how you can use functions from different modules to perform various tasks in your program.

Remember, Python comes with many built-in modules, but there are thousands more you can install to extend Python's capabilities. As you grow as a programmer, you'll discover many useful modules that can help you write better, more efficient code.

Built-in Modules

Think of Python as a toolbox. So far, you've been using the basic tools that come right on top. But there's a whole set of specialized tools built into Python that you might not have discovered yet. These are called built-in modules.

Python's Standard Library: Python comes with a vast collection of pre-written code called the Standard Library. It's like having a huge library of books that you can use anytime, without having to write them yourself!

Before we jump into specific modules, let's learn how to explore what's available. The `dir()` function shows you all the names in a module. It's like looking at a table of contents.

```
import math
print(dir(math))
```

This will print out all the functions and constants available in the `math` module. The `help()` function, as you know, gives you detailed information about a module or a specific function.

Commonly used build-in modules

The `math` module provides mathematical functions.

```
import math

# Calculate the square root
print(math.sqrt(16)) # Output: 4.0

# Calculate the sine of 45 degrees (remember to convert
to radians)
print(math.sin(math.radians(45))) # Output:
0.7071067811865476
```

The `random` module helps generate random numbers.

```
import random

# Generate a random number between 1 and 10
print(random.randint(1, 10))

# Choose a random item from a list
fruits = ['apple', 'banana', 'cherry']
print(random.choice(fruits))
```

The `datetime` module helps work with dates and times.

```
import datetime

# Get current date and time
now = datetime.datetime.now()
print(f"Current date and time: {now}")

# Create a specific date
birthday = datetime.date(1990, 5, 15)
print(f"Birthday: {birthday}")
```

Remember, these modules are just the tip of the iceberg. Python has a rich ecosystem of built-in and third-party modules that can help you tackle almost any programming task.

Introduction to Packages

Now, let's talk about packages. Think of packages as folders for your modules. A package is a collection of modules organized in a directory hierarchy. They help us organize related modules into a structure. It's like having folders and subfolders for your files, but for Python code.

Here's a simple example of what a package structure might look like:

```
mypackage/
  __init__.py
  module1.py
  module2.py
  subpackage/
    __init__.py
    module3.py
    module4.py
```

In this structure:

- █ `mypackage` is the main package
- █ `module1.py` and `module2.py` are modules in the main package
- █ `subpackage` is a package within `mypackage`
- █ `module3.py` and `module4.py` are modules in the `subpackage`

The `__init__.py` files are usually empty. They tell Python that the directory should be treated as a package.

To use modules from this package, you'd do something like this:

```
from mypackage import module1
from mypackage.subpackage import module3

module1.some_function()
module3.another_function()
```

Let's say you're building a game. You might have a package structure like this:

```
mygame/
  __init__.py
  characters.py
  items.py
  levels/
    __init__.py
    level1.py
    level2.py
  utils/
    __init__.py
    graphics.py
    sound.py
```

This structure helps keep your code organized and makes it easier to find and manage different parts of your game. When you're just starting, you don't need to create complex package structures. As your projects grow, you'll naturally start to see where packages can help organize your code better.

Remember, the goal of packages is to make your code more organized and easier to manage. Don't overcomplicate things if you don't need to!

Virtual Environments

Imagine you're working on multiple Python projects. Each project might need different versions of certain packages or even different versions of Python. This is where virtual environments come in handy.

A virtual environment is like a separate, isolated playground for each of your Python projects. It allows you to keep the dependencies for different projects separate and organized.

Here's how to create and use a virtual environment:

- 🟢 Open your command prompt or terminal.
- 🟢 Navigate to your project folder.
- 🟢 Create a virtual environment by typing:

```
python -m venv myenv
```

This creates a new virtual environment named "myenv". You can call this whatever you want, as long as it's a legal folder name. To activate the virtual environment:

- 🟢 On Windows, type: `myenv\Scripts\activate`
- 🟢 On macOS and Linux, type: `source myenv/bin/activate`

You'll see the name of your virtual environment in your command prompt, indicating it's active. Now,

any packages you install will only be available in this environment.

To deactivate the virtual environment, simply type:

deactivate

🔥 It's a good practice to create a new virtual environment for each new Python project you start. This keeps your projects clean and avoids conflicts between package versions.

Python Package Index (PyPI)

PyPI is like a massive online library of Python packages. It's where developers share their code for others to use, and where you can find packages to enhance your Python projects.

- 🟢 Finding packages: You can browse PyPI at <https://pypi.org/>. Here, you can search for packages by name or keyword.
- 🟢 Installing packages: Once you've found a package you want to use, you can install it using pip, Python's package installer.

```
pip install package_name
```

For Mac/Linux, use:

```
pip3 install package_name
```

- 🟢 Using installed packages: After installation, you can import and use the package in your Python code as we've seen before.
- 🟢 Uploading your own packages: As you become more advanced, you might want to share your own code on PyPI. This involves creating a distribution package and using tools like `'twine'` to upload it. We won't go into detail here, but it's something to look forward to as you progress!

Virtual environments and PyPI are powerful tools that will make your Python journey much smoother. They help you manage dependencies, keep your projects organized, and tap into the vast world of Python libraries.



Files

In the world of computers, a file is simply a collection of data stored on your computer. Think of it like a digital container that holds information. This could be text, images, music, or any other type of data. When we work with files in Python, we're basically reading from or writing to these containers.

When we want to work with a file, we need to tell the computer where to find it. This is something called a file path because it's the path through your folders and files that the computer needs to take to find your file.

An absolute path gives the complete route to a file from the root directory of your computer. It's like giving someone your full address, starting from the country.

Example (on Windows):

```
file_path = "C:\\Users\\YourName\\Documents\\my_file.txt"
```

Example (on Mac/Linux):

```
file_path = "/home/YourName/Documents/my_file.txt"
```

A relative path, on the other hand, gives the location of a file relative to your current working directory. It's like giving directions from where you are now.

```
file_path = "my_file.txt" # If the file is in the current directory
file_path = "data/my_file.txt" # If the file is in a subdirectory named 'data'
```

Files can be broadly categorized into two types:

- Text files: These contain human-readable characters. Examples include .txt, .csv, or .py files.
- Binary files: These contain data in a format that's not directly readable by humans. Examples include image files (.jpg, .png), audio files (.mp3), or compiled programs (.exe).

File extensions are the part of the filename after the dot (.). They help identify the type of file and the program needed to open it. For example:

- .txt for text files
- .py for Python scripts
- .jpg or .png for image files
- .csv for comma-separated value files

Python itself doesn't require file extensions to work with files, but they're useful for humans and some operating systems.

File Operations

To open a file in Python, we use the built-in `open()` function. This function takes two main arguments: the name of the file you want to open, and the mode in which you want to open it.

Here's a simple example:

```
file = open("example.txt", "r")
```

In this line, we're opening a file named 'example.txt' in read mode, which is indicated by the 'r'. The `open()` function returns a file object, which we're storing in the variable called `file`.

After we're done working with a file, it's very important to close it. This frees up system resources and ensures that all your changes are saved.

To close a file, we use the `close()` method:

```
file.close()
```

Always remember to close your files! If you forget, it can lead to problems like data loss or too many open files.

Now, here's a neat trick that Python provides: the `with` statement. This automatically closes the file for you when you're done with it. It's a safer and more convenient way to work with files.

Here's how it looks:

```
with open("example.txt", "r") as file:
    content = file.read()
    print(content)
```

In this example, we open the file, read its contents, and print them. Once the indented block is finished, Python automatically closes the file for us. Cool or what?

Let's say you're building a program that needs to read a list of names from a file. You could do it like this:

```
with open("names.txt", "r") as names_file:
    for name in names_file:
        print("Hello, " + name.strip() + "!")
```

This script opens a file called "names.txt", reads each line (which we assume is a name), and prints a greeting for each name.

File Modes

When we work with files in Python, we need to specify how we want to interact with them. This is done using file modes. Let's look at the most common ones:

■ **Read mode ('r'):** This mode is used when you want to read from a file. It's the default mode if you don't specify one.

```
file = open('example.txt', 'r')
content = file.read()
file.close()
print(content)
```

■ **Write mode ('w'):** Use this mode when you want to write to a file. Be careful! This mode will overwrite the existing content of the file.

```
file = open('example.txt', 'w')
file.write("Hello, World!")
file.close()
```

■ **Append mode ('a'):** This mode is used to add content to the end of a file without overwriting existing content.

```
file = open('example.txt', 'a')
file.write("\nThis is a new line.")
file.close()
```

■ **Binary mode ('b'):** Use this mode when working with non-text files like images or audio files. You'll often see it combined with other modes like 'rb' or 'wb'.

```
file = open('image.jpg', 'rb')
content = file.read()
file.close()
```

■ **Read and write mode ('r+'):** This mode allows you to both read from and write to a file.

```
file = open('example.txt', 'r+')
content = file.read()
file.write("\nAdding this line.")
file.close()
```

Always remember to close your files after you're done working with them!

Reading Files

Now that we know how to open files, let's look at different ways to read their content:

The `read()` method reads the entire content of the file as a single string.

```
file = open('example.txt', 'r')
content = file.read()
file.close()
print(content)
```

On the other hand, the `readline()` method reads one line at a time from the file.

```
file = open('example.txt', 'r')
first_line = file.readline()
second_line = file.readline()
file.close()
print(first_line)
print(second_line)
```

We can use `readlines()` to read all lines from the file and return them as a list of strings.

```
file = open('example.txt', 'r')
lines = file.readlines()
file.close()
for line in lines:
    print(line.strip()) # strip() removes the newline
                        # character
```

You can also read a file line by line using a for loop, which is often the most efficient way.

```
file = open('example.txt', 'r')
for line in file:
    print(line.strip())
file.close()
```

Sometimes you might want to read only a certain number of characters. You can do this by passing an argument to the `read()` method.

```
file = open('example.txt', 'r')
first_10_chars = file.read(10)
file.close()
print(first_10_chars)
```

Writing Files

Just as you'll want to open and read files, you'll also want to save your data back to files for you to

distribute or to use later. Python makes this easy with two main methods: `write()` and `writelines()`.

The `write()` method allows you to add a single string to a file.

```
# Open a file in write mode
with open("example.txt", "w") as file:
    file.write("Hello, Python!")
```

Here, we're opening a file named "example.txt" in write mode (that's what the "w" means). Then we use the `write()` method to add the text "Hello, Python!" to the file.

Using "w" mode will overwrite any existing content in the file. If you want to add to the file without erasing what's already there, use "a" for append mode instead.

If you have multiple lines to write, you can use the `writelines()` method. It takes a list of strings and writes them to the file:

```
# Open a file in write mode
with open("shopping_list.txt", "w") as file:
    items = ["Apples\n", "Bananas\n", "Milk\n"]
    file.writelines(items)
```

This will create a file called "shopping_list.txt" with each item on a new line.

Notice in the previous example, we added "\n" at the end of each item. This creates a new line after each item. Without it, everything would be on one line.

File Positions

When you're working with files, Python keeps track of where you are in the file, kind of like a bookmark. You can check this position and even move it around.

The `tell()` method tells you your current position in the file.

```
with open("example.txt", "r") as file:
    print(file.tell()) # This will print 0, as we're at
                      # the start
    file.read(5) # Read 5 characters
    print(file.tell()) # This will print 5
```

The `seek()` method lets you move to a specific position in the file:

```
with open("example.txt", "r") as file:
    file.seek(5) # Move to the 6th character (remember,
                # we start counting at 0)
    print(file.read()) # Read from the 6th character to
                      # the end
```

You can use `seek()` to move around in a file. This is useful for reading specific parts of a file or updating certain sections:

```
with open("example.txt", "r+") as file: # r+ mode
    allows reading and writing
    file.seek(7) # Move to the 8th character
    file.write("Python is awesome!")
    # Overwrite from this position
```

Be careful when using `seek()` with write operations. It's easy to accidentally overwrite data if you're not in the right position!

Working with CSV Files

CSV (Comma-Separated Values) files are a common way to store tabular data. Python's csv module makes it easy to work with these files.

First, let's read a simple CSV file (you can find this file in the supplementary information):

```
import csv

with open('example.csv', 'r') as file:
    csv_reader = csv.reader(file)
    for row in csv_reader:
        print(row)
```

This code opens the file, creates a csv reader object, and then prints each row.

Now, let's write some data to a CSV file:

```
import csv

data = [['Name', 'Age'], ['Alice', 30], ['Bob', 25]]

with open('new_file.csv', 'w', newline='') as file:
    csv_writer = csv.writer(file)
    csv_writer.writerows(data)
```

This creates a new CSV file and writes our data to it. You can now load and save your own data. Amazing!

Working with JSON Files

To work with JSON in Python, we'll use the `json` module. This module comes built-in with Python, so you don't need to install anything extra.

Let's start by importing the module:

```
import json
```

Now, let's look at how to read JSON data. There are two main ways to do this:

- Reading from a file using `json.load()`
- Reading from a string using `json.loads()`
(notice the 's' at the end)

Let's say we have a JSON file called "data.json" with the following content:

```
{
  "name": "Alice",
  "age": 30,
  "city": "New York"
}
```

To read this file, we can use `json.load()` like this:

```
with open('data.json', 'r') as file:
    data = json.load(file)

print(data)
print(f"Name: {data['name']}")
print(f"Age: {data['age']}")
print(f"City: {data['city']}")
```

This will print:

```
{'name': 'Alice', 'age': 30, 'city': 'New York'}
Name: Alice
Age: 30
City: New York
```

If you have JSON data as a string instead of a file, you can use `json.loads()`:

```
json_string = '{"name": "Bob", "age": 25, "city": "London"}'
data = json.loads(json_string)

print(data)
print(f"Name: {data['name']}")
```

Can you spot the similarity to dictionaries yet? Let's look at writing JSON data. Again, we have two main methods:

1. Writing to a file using `json.dump()`
2. Converting to a JSON string using `json.dumps()`
(notice the 's' at the end again)

To write data to a JSON file:

```
data = {
  "name": "Charlie",
  "age": 35,
  "city": "Paris"
}

with open('new_data.json', 'w') as file:
    json.dump(data, file, indent=4)
```

This will create a file called "new_data.json" with nicely formatted JSON data. The `indent=4` argument makes the output more readable by adding indentation.

If you want to convert your data to a JSON string instead of writing to a file, use `json.dumps()`:

```
data = {"name": "David", "age": 40, "city": "Tokyo"}
json_string = json.dumps(data)
print(json_string)
```

This will print:

```
{"name": "David", "age": 40, "city": "Tokyo"}
```

JSON is great for storing configuration settings, saving application data, or exchanging information between different systems. For example, many web application interfaces (APIs) send and receive data in JSON format.

File System Operations (optional)

Python allows you to work with the files on your computer directly, such as creating new files, deleting existing ones or even moving files around. For this we'll use the `os` module: Your Swiss Army Knife for File Operations.

First things first, let's import the `os` module. This module gives us lots of useful functions to work with files and directories.

```
import os
```

Think of the '`os`' module as a toolbox full of different tools for working with your computer's files and folders.

Sometimes you need to create new folders in your programs. Python makes this easy with two functions:

- `os.mkdir()`: This creates a single new directory.

```
os.mkdir("new_folder")
```

This will create a folder named "new_folder" in your current directory.

- `os.makedirs()`: This can create multiple nested directories at once.

```
os.makedirs("parent_folder/child_folder/grandchild_folder")
```

This creates a folder structure like this: parent_folder › child_folder › grandchild_folder. If you try to create a folder that already exists, you'll get an error.

We can also see what's inside folders and for this we also have different methods. `os.listdir()` gives you

a list of all files and folders in a directory. The example below will print a list of all items in "my_folder".

```
contents = os.listdir("my_folder")
print(contents)
```

`os.scandir()`: This is a more advanced way to list directory contents, giving you more information about each item. This does the same thing as `listdir()`, but it's more efficient for large directories. You could use this to build a simple file explorer or to find all image files in a folder.

```
with os.scandir("my_folder") as entries:
    for entry in entries:
        print(entry.name)
```

Before we work with a file, it's often good to check if it exists:

```
if os.path.exists("my_file.txt"):
    print("The file exists!")
else:
    print("The file does not exist.")
```

This is really useful to avoid errors when trying to open or modify files.

Lastly, let's learn how to get detailed information about a file:

```
file_info = os.stat("my_file.txt")
print(f"Size: {file_info.st_size} bytes")
print(f>Last modified: {file_info.st_mtime}")
```

The `stat()` function gives you information like the file's size, when it was last modified, and more.

Practical use: You could use this to find large files taking up space, or to check when a file was last updated.

Working with Paths

When dealing with files, we often need to work with file paths as we've seen. Python's `os.path` module provides useful functions for this purpose. The `os.path` module helps us manipulate file paths in a way that works across different operating systems.

```
import os.path
```

To create file paths, we use `os.path.join()`. This function combines path components intelligently.

```
folder = "documents"
filename = "report.txt"
full_path = os.path.join(folder, filename)
print(full_path)
```

This method is better than manually combining strings because it handles the correct separator for your operating system.

To extract parts of a path, we use `os.path.basename()` and `os.path.dirname()`:

```
path = "/home/user/documents/report.txt"
file_name = os.path.basename(path)
dir_name = os.path.dirname(path)

print(file_name) # Output: report.txt
print(dir_name) # Output: /home/user/documents
```

`os.path.basename()` takes a file path and returns just the name of the file at the end, while `os.path.dirname()` returns the directory path without the file name. These functions help you easily extract specific parts of a file path, which is useful when working with files and directories in your programs.

File Encoding

Now, let's talk about file encoding, which is how computers store text in files. Character encoding is a way to represent text in computer files. Common encodings include:

- 🟩 UTF-8: Can represent any character in the Unicode standard (recommended for most cases).
- 🟩 ASCII: Basic encoding for English text (limited to 128 characters).

When opening a file, you can specify the encoding:

```
with open("myfile.txt", "r", encoding="utf-8") as file:
    content = file.read()
    print(content)
```

File Compression

Python lets us work directly with compressed files or compress and uncompress them ourselves. Compressed files take up less space on your computer, which is great for storage and sharing. Python has built-in modules to help us work with these files.

The `zipfile` module lets us create, read, write, and list zip files.

```
import zipfile

# Creating a zip file
with zipfile.ZipFile('my_zip.zip', 'w') as my_zip:
    my_zip.write('file1.txt')
    my_zip.write('file2.txt')

# Reading from a zip file
with zipfile.ZipFile('my_zip.zip', 'r') as my_zip:
    my_zip.extractall('extracted_folder')
```

This code creates a zip file with two files in it, then extracts all the contents to a new folder.



Error Handling

We've mentioned throwing and handling errors many times up to now without learning about what that actually means. Well, it's about to change!

Imagine you're cooking a new recipe. Sometimes things don't go as planned - you might add too much salt or forget an ingredient. In programming, similar things can happen. Error handling is like having a plan B for when things go wrong in your code.

So, what exactly is error handling? It's the process of anticipating, detecting, and resolving errors that can occur while your program is running. These errors could be mistakes in your code, unexpected user inputs, or even problems with your computer's resources.

Why is error handling important?

- It helps your program continue running even when problems occur.
- It improves the user experience by providing helpful error messages instead of crashing.
- It makes your code more robust and reliable.

Exception Handling: try-catch block

Exception handling is a way to deal with errors that might happen while your program is running, so your code doesn't just crash when something goes wrong.

It is like having a safety net for your code. It's a way to catch errors, or "exceptions," that happen while your program is running. Instead of letting these errors stop your program, you can decide what to do when they occur.

Try-Except Blocks

The main tool for handling exceptions in Python is the **try-except** block. The basic syntax for the **try-except** block is:

```
try:
    # Code that might raise an exception except
    SomeException as e:
    # Code to handle the exception
finally:
    # Code that will run no matter what
```

The **try-except** block, the code within the **try** block is executed, and if an exception occurs, the code in the **except** block is run to handle the error. Here's an example:

```
try:
    # Code that might cause an error
    result = 10 / 0
except ZeroDivisionError:
    # What to do if a specific error occurs
    print("Oops! You can't divide by zero.")
```

In this example, we're trying to divide 10 by 0, which would normally cause an error. But because we've put it in a **try** block and caught the specific error (**ZeroDivisionError**) in the **except** block, our program doesn't crash. Instead, it prints a friendly message.

Multiple Except Blocks

Sometimes, different types of errors can occur in your code. You can handle these with multiple **except** blocks:

```
try:
    number = int(input("Enter a number: "))
    result = 10 / number
except ValueError:
    print("That's not a valid number!")
except ZeroDivisionError:
    print("You can't divide by zero!")
```

This code can handle two types of errors: if the user doesn't enter a valid number, or if they enter zero.

The Finally Block

The **finally** block is a place to put code that you want to run no matter what, whether an error occurred or not:

```
try:
    file = open("example.txt", "r")
    # Do something with the file
except FileNotFoundError:
    print("The file doesn't exist!")
finally:
    file.close() # This will run every time
```

This is particularly useful for things like closing files or network connections.

Raising Exceptions

Sometimes, you might want to cause an exception yourself. You can do this with the `raise` keyword. In this example, for example, we're creating our own error when someone tries to set a negative age.

```
def set_age(age):
    if age < 0:
        raise ValueError("Age cannot be negative")
    print(f"Age set to {age}")

try:
    set_age(-5)
except ValueError as error:
    print(error)
```

Error Handling Best Practices

Let's start with an important principle: "Fail fast and fail loudly." This means that when something goes wrong in your program, it's better to have it crash immediately and give you a clear error message, rather than continuing to run with incorrect data.

When handling errors, it's best to catch specific types of exceptions rather than using a catch-all approach. This allows you to handle different errors in different ways.

```
try:
    number = int(input("Enter a number: "))
    result = 10 / number
except ValueError:
    print("That's not a valid number!")
except ZeroDivisionError:
    print("You can't divide by zero!")
except Exception as e:
    print(f"An unexpected error occurred: {e}")
```

Here, we're handling different types of errors separately. This makes our code more precise and easier to debug.

An empty catch block (one that doesn't do anything when it catches an error) can hide problems in your code. Always include some kind of handling or logging in your catch blocks. It can be as simple as printing the error to screen.

Bad example (avoid this):

```
try:
    # Some risky operation
    risky_function()
except Exception:
    pass # This is bad! Don't do this!
```

Good example:

```
try:
    risky_function()
except Exception as e:
    print(f"An error occurred: {e}")
    # You might also want to log the error or take some
    # corrective action
```

When an error occurs, it's crucial to log not just the error message, but also relevant information about the state of your program. This will make debugging much easier.

We can use the `logging` module to format the way we log errors.

```
import logging

logging.basicConfig(level=logging.INFO)

def process_data(data):
    try:
        result = perform_calculation(data)
        return result
    except Exception as e:
        logging.error(f"Error processing data: {e}")
        logging.error(f"Input data was: {data}")
        raise # Re-raise the exception after logging
```

In this example, we're logging both the error and the input data that caused the error. This gives us valuable context for debugging.

Error Messages and User Communication

When an error occurs, it's important to let the user know what happened in a way they can understand.

```
def divide_numbers(a, b):
    if b == 0:
        print("Error: Cannot divide by zero!")
    else:
        result = a / b
        print(f"The result is: {result}")

# Example usage
divide_numbers(10, 0)
```

In this example, we check if the second number is zero before performing the division. If it is, we print a clear message explaining the problem.

When an error occurs, it's helpful to tell users what they can do to fix the problem. Let's enhance our division example:

```
def divide_numbers(a, b):
    if b == 0:
        print("Error: Cannot divide by zero!")
        print("Please try again with a non-zero second number.")
    else:
        result = a / b
        print(f"The result is: {result}")

# Example usage
divide_numbers(10, 0)
```

It's important to strike a balance between providing enough information and not overwhelming the user with technical jargon.

```
def read_file(filename):
    try:
        with open(filename, 'r') as file:
            content = file.read()
            print(content)
    except FileNotFoundError:
        print(f"Error: The file '{filename}' was not found.")
        print("Please check if the file exists and try again.")
    except PermissionError:
        print(f"Error: You don't have permission to read '{filename}'.")
        print("Please check your file permissions and try again.")

read_file("nonexistent_file.txt")
```

Here we're catching specific errors and providing user-friendly explanations without delving into technical details of the file system.

Debugging Techniques

Debuggers are tools that help you examine your code as it runs, allowing you to pause execution and inspect variables. Python comes with a built-in debugger called `pdb`.

```
import pdb

def add_numbers(a, b):
    result = a + b
    pdb.set_trace() # This line starts the debugger
    return result

print(add_numbers(3, 4))
```

When you run this code, it will pause at the `pdb.set_trace()` line. You can then use commands:

- ▣ `'n'` to go to the next line
- ▣ `'p variable_name'` to print a variable's value
- ▣ `'c'` to continue execution

This allows you to check if your variables have the expected values and helps identify where things might be going wrong. This form of debugging and

going through your program step by step is extremely useful when things aren't going as expected.

When an error occurs, Python provides a stack trace - a report of the active function calls at the point where the error happened.

```
def divide_numbers(a, b):
    return a / b

def calculate():
    result = divide_numbers(10, 0)
    print(result)

calculate()
```

This will produce an error message (stack trace) like:

```
Traceback (most recent call last):
  File "example.py", line 7, in <module>
    calculate()
  File "example.py", line 5, in calculate
    result = divide_numbers(10, 0)
  File "example.py", line 2, in divide_numbers
    return a / b
ZeroDivisionError: division by zero
```

Reading from bottom to top:

- ▣ We see the specific error (**ZeroDivisionError**)
- ▣ We can trace back through the function calls to see where the error originated

This helps you pinpoint exactly where the problem occurred in your code.

To fix a bug, you first need to be able to make it happen consistently. Here are some tips:

- ▣ Keep track of the exact steps that lead to the error
- ▣ Try to create a minimal example that still produces the error
- ▣ Use print statements to log variable values at different points in your code

```
def process_data(data):
    print(f"Received data: {data}") # Log the input
    result = data * 2
    print(f"Processed result: {result}") # Log the output
    return result

# Test with different inputs
print(process_data(5))
print(process_data("hello")) # This will cause an error
```

By adding print statements, you can see exactly what's happening at each step, making it easier to identify where things go wrong.

Testing Error Handling Code

When we write code to handle errors, we need to make sure it works correctly. This is where testing comes in handy.

Unit tests are small, focused tests that check if a specific part of your code works as expected. When it comes to error handling, we want to test if our code correctly catches and handles exceptions.

```
def divide(a, b):
    if b == 0:
        raise ValueError("Cannot divide by zero")
    return a / b

# Unit test
def test_divide_by_zero():
    try:
        divide(10, 0)
    except ValueError as e:
        assert str(e) == "Cannot divide by zero"
    else:
        assert False, "Expected ValueError"

# Run the test
test_divide_by_zero()
print("Test passed!")
```

In this example, we're testing if our divide function correctly raises a `ValueError` when we try to divide by zero. We will look at the `assert` command later in the course.

Play around with raising and catching errors in your code. Being able to handle errors properly will make your code more stable and avoid crashes down the line.



Object-oriented programming

In this chapter, we're going to explore a powerful programming paradigm that will change the way you think about structuring your code.

Object-oriented programming (OOP) is a programming paradigm - that's just a fancy way of saying it's a style or approach to writing code. This particular style is based on the concept of "objects".

You can think of an object as a container that holds both data and the functions that operate on that data all in the same place. It's like a digital representation of a real-world entity. For example, if you were programming a game, you might have a "Player" object that contains data like the player's name and score, as well as functions like "move" or "jump".

Now, you might be thinking, "This seems more complicated than just using variables and functions. Why should I use OOP?"

Great question! OOP offers several advantages:

- **Modularity:** OOP allows you to break down your code into smaller, more manageable pieces. Each object is like a mini-program on its own.
- **Reusability:** Once you've created a class, you can reuse it as many times as you want. Need 100 players? No problem! Just create 100 Player objects.

- Scalability: As your programs grow larger, OOP helps keep things organized. It's much easier to add new features or modify existing ones when your code is well-structured.
- Maintainability: When you need to fix bugs or update your program, OOP makes it easier to locate and fix issues without affecting the entire program.

While OOP is powerful, it's not always the best solution for every problem. Simple scripts or small programs might not need the overhead of OOP. As you gain more experience, you'll develop a sense for when to use OOP and when simpler approaches might suffice.

Classes

Classes are what makes object-oriented programming possible. Once you get the hang of them, you'll be using them all the time.

What is a Class? Think of a class as a blueprint or a template. Just like architects use blueprints to design buildings, we use classes to design objects in our code. A class defines what attributes (data) and methods (functions) an object will have.

To create a class in Python, we use the **class** keyword. Here's the basic syntax:

```
class ClassName:
    # Class content goes here
```

For example, let's create a simple class called Dog:

```
class Dog:
    pass # We'll add more to this soon!
```

The `pass` keyword is just a placeholder. It tells Python, "Hey, I know this class is empty right now, and that's okay!"

- Classes have two main types of members:

Attributes: These are like variables that belong to the class. They represent the data or state of an object.

Methods: These are functions that belong to the class. They define the behaviours or actions an object can perform.

Let's add some attributes and methods to our Dog class:

```
class Dog:
    # Class attribute
    species = "Canis familiaris"

    # Constructor method
    def __init__(self, name, age):
        # Instance attributes
        self.name = name
        self.age = age

    # Instance method
    def bark(self):
        return "Woof!"
```

There's a fair bit here, so let's break this down:

- species** is a class attribute. It's shared by all instances of the class.
- __init__** is a special method called a constructor. It's used to initialize new objects.
- self** refers to the instance of the class being created.
- name** and **age** are instance attributes. Each object will have its own values for these.
- bark** is an instance method. It defines a behaviour for our Dog objects.

We will look at all of these components and concepts in more detail, so don't worry if this is a lot to take in right now. To start, there are some conventions and rules about creating classes:

- Always start class names with a capital letter. This is a convention in Python.
- The **self** parameter in methods refers to the instance of the class. Don't forget to include it as the first parameter in your methods!
- You can create as many objects as you want from a single class.
- Classes help you organize your code and make it reusable. Think about how you can group related data and functions into classes.

Objects

Objects are instances of a class. If you think of a class as a blueprint or a template, an object is the actual thing created from that blueprint.

Let's use a real-world example. Imagine you have a blueprint for a house. The blueprint isn't the house itself - it's just the plan. When you actually build a house using that blueprint, you've created an instance of that house design. In programming terms, the blueprint is the **class**, and the actual house you built is the **object**.

Let's change up our Dog class that we created earlier.

```
class Dog:
    def __init__(self, name, breed):
        self.name = name
        self.breed = breed

    def bark(self):
        print(f"{self.name} says Woof!")
```

This Dog class is our blueprint. It defines what properties a dog should have (a name and a breed) and what it can do (bark).

To create an object from this class, we use a process called *instantiation*.

```
my_dog = Dog("Buddy", "Golden Retriever")
```

We've just created an object! Woof! `my_dog` is now an instance of the Dog class. We can create as many dog objects as we want:

```
another_dog = Dog("Max", "Poodle")
```

Each of these objects is independent. They're based on the same blueprint, but they're separate instances with their own data.

We can use these objects like this:

```
my_dog.bark() # Output: Buddy says Woof!
print(another_dog.breed) # Output: Poodle
```

Objects in Python go through three main stages:

1. **Creation:** This happens when we instantiate the class, like we did with `my_dog = Dog("Buddy", "Golden Retriever")`.
2. **Use:** This is when we work with the object, calling its methods or accessing its attributes.
3. **Destruction:** In Python, we don't need to manually destroy objects. Python has a system called garbage collection that automatically cleans up objects that are no longer being used.

Consider a case where you're writing a program for a veterinary clinic. You could use the Dog class to keep track of the clinic's patients:

```
patients = [
    Dog("Buddy", "Golden Retriever"),
    Dog("Max", "Poodle"),
    Dog("Charlie", "Beagle")
]

for patient in patients:
    print(f"Checking {patient.name}, a {patient.breed}")
    patient.bark()
```

This allows you to easily manage and interact with multiple dog objects in your program.

Attributes (Properties)

Attributes, also known as properties, are essentially the data associated with a class or its objects. They're like variables that belong to a class or an object. We have two main types of attributes: instance variables and class variables. Let's break these down.

■ Instance variables are unique to each object. They hold data that can be different for every instance of a class.

■ On the other hand, class variables are shared among all instances of a class - they hold data that's the same for every object.

Let's see an example:

```
class Dog:
    # This is a class variable
    species = "Canis familiaris"

    def __init__(self, name, breed):
        # These are instance variables
        self.name = name
        self.breed = breed

# Creating two dog objects
buddy = Dog("Buddy", "Golden Retriever")
max = Dog("Max", "Poodle")

print(buddy.species) # Output: Canis familiaris
print(max.species)   # Output: Canis familiaris
print(buddy.name)    # Output: Buddy
print(max.name)       # Output: Max
```

Here, `species` is a class variable - it's the same for all dogs. But `name` and `breed` are instance variables - they're different for each dog.

We define class variables directly in the class, outside of any method. Instance variables, however, are typically defined and initialized in the `__init__` method. The `__init__` method is a special method in Python classes that's called when an object is created.

```
class Cat:
    # Class variable
    species = "Felis catus"

    def __init__(self, name, colour):
        # Instance variables
        self.name = name
        self.colour = colour
```

Methods

Methods are functions that are associated with a class. Think of them as actions that objects of that class can perform. Just like how a dog can bark or a car

can accelerate, objects in our code can have their own set of actions or behaviours.

■ **Instance Methods:** These are the most common type of methods. They operate on individual instances of a class. Here's a simple example:

```
class Dog:
    def __init__(self, name):
        self.name = name

    def bark(self):
        print(f"{self.name} says: Woof!")

my_dog = Dog("Buddy")
my_dog.bark() # Output: Buddy says: Woof!
```

In this example, `bark()` is an instance method. It's called on a specific instance of the Dog class (in this case, `my_dog`).

■ **Class Methods:** These are methods that work with the class itself, rather than instances of the class. They're defined using the `@classmethod` decorator.

```
class MathOperations:
    @classmethod
    def add(cls, a, b):
        return a + b

result = MathOperations.add(5, 3)
print(result) # Output: 8
```

In this case, `add()` is a class method. We can call it on the class itself, without creating an instance.

Methods are defined inside the class, just like regular functions, but with one key difference: they always take `self` as their first parameter (for instance methods).

```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

rect = Rectangle(5, 3)
print(rect.area()) # Output: 15
```

The `self` parameter is crucial. It refers to the instance of the class that the method is being called on. This allows the method to access and modify the instance's attributes.

Methods are incredibly useful for encapsulating behaviour. For example, if you're building a banking application, you might have a `BankAccount` class with methods like `deposit()`, `withdraw()`, and `check_balance()`. This keeps all the account-related operations neatly organized within the class.

Constructors

Constructors are special methods in Python classes that initialize new objects. Think of them as the "setup" function for your objects. The main job of a constructor is to set up the initial state of an object when it's created. It's like preparing a new car before its first drive - you need to fill it with gas, adjust the mirrors, and set the seat position.

In Python, the constructor method is always named `__init__()`.

Let's look at a simple example:

```
class Car:
    def __init__(self, make, model):
        self.make = make
        self.model = model

my_car = Car("Toyota", "Corolla")
print(f"My car is a {my_car.make} {my_car.model}")
```

In this example, `__init__` is our constructor. It sets up the make and model attributes for each new Car object. Remember the Dog constructor?

Default constructors are constructors that takes no arguments (except `self`).

```
class SimpleCar:
    def __init__(self):
        self.make = "Unknown"
        self.model = "Unknown"

basic_car = SimpleCar()
print(f"Basic car: {basic_car.make} {basic_car.model}")
```

Parameterized constructors, on the other hand, accept arguments to initialize the object with specific values.

```
class DetailedCar:
    def __init__(self, make, model, year, colour):
        self.make = make
        self.model = model
        self.year = year
        self.colour = colour

my_detailed_car = DetailedCar("Honda", "Civic", 2022, "Blue")
print(f"My car: {my_detailed_car.year} {my_detailed_car.colour} {my_detailed_car.make} {my_detailed_car.model}")
```

Destructors in Python

In Python, destructors are methods called when an object is about to be destroyed. They're used for cleanup operations. They are used to perform any necessary cleanup when an object is no longer needed.

This might include closing files, network connections, or freeing up system resources.

The destructor method is named `__del__()`.

```
class FileHandler:
    def __init__(self, filename):
        self.filename = filename
        self.file = open(filename, "w")
        print(f"File {self.filename} opened.")

    def __del__(self):
        self.file.close()
        print(f"File {self.filename} closed.")

# Using the FileHandler
handler = FileHandler("example.txt")
# Some operations...
del handler # This will trigger the __del__ method
```

Python uses automatic garbage collection, which means you usually don't need to worry about manually freeing memory, like you do in languages like C++. The Python interpreter keeps track of object references and automatically frees the memory when an object is no longer in use.

However, for resources other than memory (like file handles or network sockets), it's good practice to close them explicitly when you're done, rather than relying solely on the destructor.

Encapsulation

Encapsulation and Abstraction are principles that help us write cleaner, more organized, and more maintainable code. Let's look at them in turn.

Encapsulation is like wrapping up your data and the methods that work on that data into a neat little package. Imagine you have a present - the box is the class, and inside are the gift (the data) and instructions on how to use it (the methods).

The main idea behind encapsulation is to protect our data from unauthorized access or modification. We achieve this through what we call "information hiding".

Here's a simple example:

```
class BankAccount:
    def __init__(self):
        self.__balance = 0 # This is a private attribute

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount

    def get_balance(self):
        return self.__balance
```

In this example, we've created a `BankAccount` class. The balance is stored in a private attribute `__balance`. We can't directly access or modify this balance from outside the class. Instead, we use methods like `deposit` and `get_balance` to interact with it.

Python doesn't have strict access modifiers like some other languages, and technically we could modify `__balance` directly, but we use conventions. A single underscore `_` suggests protected (accessible within the class and its subclasses), and double underscores `__` indicate private (only accessible within the class).

Abstraction

Abstraction is about simplifying complex reality by modelling classes based on the essential properties and behaviours. It's like driving a car - you don't need to know how the engine works; you just need to know how to use the steering wheel and pedals.

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius ** 2
```

In this example, we've created an abstract base class `Shape` with an abstract method `area`. The `Circle` class inherits (more on this in the next chapter) from `Shape` and implements the `area` method.

Abstract classes and interfaces (which are similar concepts) allow us to define a common structure that multiple classes can follow, without implementing all the details.

Remember, these concepts might take some time to fully grasp, but they're incredibly powerful once you get the hang of them. Practice writing your own classes and think about how you can apply these principles to make your code better.

Basic Class Relationships

Understanding how different classes can interact and connect with each other is crucial for designing efficient and logical programs.

Association is the simplest form of relationship between classes. It means that objects of one class are somehow connected to objects of another class. Think

of it as a loose connection or a "uses" relationship. For example, let's consider Student and Course classes.

```
class Student:
    def __init__(self, name):
        self.name = name
        self.courses = []

    def enroll(self, course):
        self.courses.append(course)

class Course:
    def __init__(self, name):
        self.name = name

# Usage
student = Student("Alice")
course = Course("Python 101")
student.enroll(course)
```

Here, a **Student** can **enroll** in multiple **Courses**, and a **Course** can have multiple **Students**. They're associated, but neither owns the other.

Aggregation is a bit stronger than association. It's often described as a "has-a" relationship, where one class contains objects of another class. The key point is that the contained object can exist independently.

```
class Employee:
    def __init__(self, name):
        self.name = name

class Department:
    def __init__(self, name):
        self.name = name
        self.employees = []

    def add_employee(self, employee):
        self.employees.append(employee)

emp1 = Employee("Bob")
emp2 = Employee("Charlie")
dept = Department("IT")
dept.add_employee(emp1)
dept.add_employee(emp2)
```

In this case, a **Department** has **Employees**, but **Employees** can exist without a **Department**.

Finally, composition is the strongest form of relationship between classes. It's a special case of aggregation where the contained object cannot exist independently of the container. It's often described as a "part-of" relationship.

```
class Engine:
    def __init__(self, type):
        self.type = type

class Car:
    def __init__(self, model):
        self.model = model
        self.engine = Engine("V6") # The Engine is
        # created when the Car is created

# Usage
car = Car("Sedan")
print(car.engine.type) # Output: V6
```

In this case, the Engine is a part of the Car and doesn't exist independently. When the Car object is destroyed, the Engine object is also destroyed.

'self' Keyword (advanced)

The 'self' keyword is a crucial part of object-oriented programming in Python. It's used within class methods to refer to the instance of the class that's calling the method. In simpler terms, 'self' helps Python understand which specific object we're working with.

```
class Dog:
    def __init__(self, name):
        self.name = name

    def bark(self):
        print(f"{self.name} says: Woof!")

my_dog = Dog("Buddy")
my_dog.bark() # Output: Buddy says: Woof!
```

In this example, **'self'** is used in both the **'init'** method (which is called when we create a new Dog) and the **'bark'** method. It allows us to access and modify the object's attributes, like **'name'** in this case.

The **'self'** keyword is particularly useful when we need to distinguish between instance variables (attributes of the object) and parameters passed to a method.

```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def set_dimensions(self, width, height):
        self.width = width
        self.height = height

rect = Rectangle(5, 10)
rect.set_dimensions(7, 14)
```

In the **'set_dimensions'** method, **'self.width'** refers to the instance variable, while **'width'** (without 'self') refers to the parameter passed to the method.

Always remember to include **'self'** as the first parameter in your class methods. Python will automatically pass the instance when you call the method, so you don't need to pass 'self' explicitly when calling the method.

Object Comparison

When we're dealing with objects, we often need to compare them. But how do we do that? There are two main ways:

■ Using the `==` operator: The `==` operator checks if two objects are the same object in memory.

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

person1 = Person("Alice", 30)
person2 = Person("Alice", 30)
person3 = person1

print(person1 == person2) # False
print(person1 == person3) # True
```

Even though `person1` and `person2` have the same content, they're different objects in memory, so `==` returns False. But `person3` is the same object as `person1`, so `==` returns True.

■ Often, we want to check if two objects have the same content, not if they're the same object. For this, we can implement a custom method:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __eq__(self, other):
        if isinstance(other, Person):
            return self.name == other.name and self.age == other.age
        return False

person1 = Person("Alice", 30)
person2 = Person("Alice", 30)

print(person1 == person2) # True
```

By implementing the `__eq__` method, we can define what it means for two `Person` objects to be equal.

Object Serialization

Object Serialization is a fancy way of saying "converting objects into a format that can be easily stored or sent around." Think of it like packing your suitcase for a trip. You're taking all your stuff (the object) and organizing it in a way that makes it easy to transport (serialization).

In Python, there are a few basic ways to do this. One common method is using the `pickle` module. Here's a simple example:

```
import pickle

class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age

# Create a Dog object
my_dog = Dog("Buddy", 5)
```

```
# Serialize the object
with open('my_dog.pkl', 'wb') as file:
    pickle.dump(my_dog, file)

# Later, we can deserialize it
with open('my_dog.pkl', 'rb') as file:
    loaded_dog = pickle.load(file)

print(f"Loaded dog: {loaded_dog.name}, {loaded_dog.age} years old")
```

In this example, we're "pickling" our `Dog` object, saving it to a file, and then "unpickling" it later. This is super useful when you want to save the state of your program or send data between different parts of your application.

A practical use case might be saving user preferences in a game or application. You could serialize the user's settings object and save it to a file, then load it back when the user starts the app again.

⚠ While `pickle` is convenient, it's not secure against malicious data. Don't use it with data from untrusted sources!

Best Practices

Let's begin with some important best practices that will help you write cleaner, more maintainable code. We follow specific naming conventions for classes, methods, and attributes:

- Class names use CamelCase: Each word starts with a capital letter, with no underscores.
- Method and attribute names use snake_case: All lowercase with underscores between words.

```
class BankAccount:
    def __init__(self, account_number, balance):
        self.account_number = account_number
        self.balance = balance

    def deposit_money(self, amount):
        self.balance += amount
```

In this example, `BankAccount` is our class name in CamelCase, while `account_number`, `balance`, and `deposit_money` use snake_case.

The single responsibility principle states that a class should have only one reason to change. In other words, a class should have a single, well-defined purpose. For example, instead of having a `User` class that handles both user authentication and email sending, we could split it into two classes:

```
class User:
    def __init__(self, username, password):
        self.username = username
        self.password = password
```

```
def authenticate(self):
    # Authentication logic here
    pass

class EmailSender:
    def send_email(self, user, message):
        # Email sending logic here
        pass
```

Keeping Classes focused and cohesive is closely related to the Single Responsibility Principle. A cohesive class is one where all its methods and attributes are closely related to the class's main purpose.

Common Design Patterns

Design patterns are reusable solutions to common programming problems. Let's look at two popular patterns:

■ Singleton Pattern

The Singleton pattern ensures a class has only one instance and provides a global point of access to it. This can be useful for managing shared resources.

```
class Singleton:
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
        return cls._instance

# Usage
s1 = Singleton()
s2 = Singleton()
print(s1 is s2) # Output: True
```

■ Factory Pattern

The Factory pattern provides an interface for creating objects without specifying their exact class. This can be useful when the exact type of object needed isn't known until runtime.

Here's a basic example:

```
class Dog:
    def speak(self):
        return "Woof!"

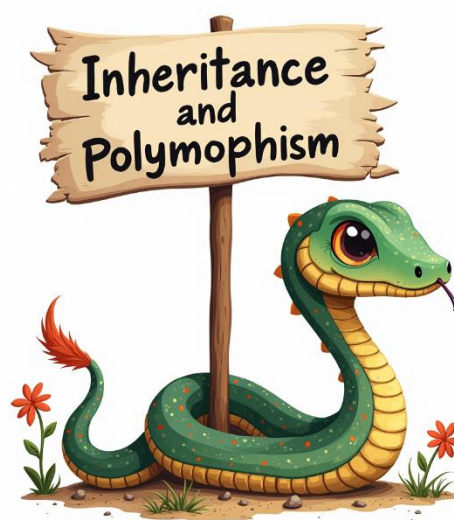
class Cat:
    def speak(self):
        return "Meow!"

class AnimalFactory:
    def create_animal(self, animal_type):
        if animal_type == "dog":
            return Dog()
        elif animal_type == "cat":
            return Cat()

# Usage
factory = AnimalFactory()
dog = factory.create_animal("dog")
cat = factory.create_animal("cat")
```

```
print(dog.speak()) # Output: Woof!
print(cat.speak()) # Output: Meow!
```

There's a lot to take in when it comes to object-oriented programming. Take your time to fully understand the principles and play around making your own classes and objects. Once the core principles become second nature, you'll start wondering how you've been able to program without objects so far!



Inheritance and Polymorphism

Inheritance is a fundamental principle in object-oriented programming. It's a mechanism that allows us to create new classes based on existing classes. This means we can build on what we already have, saving time and reducing code duplication.

Let's break it down with some key terms:

- Superclass: This is the original class from which we're inheriting. You might also hear it called the parent class or base class.
- Subclass: This is the new class we create that inherits from the superclass. It's sometimes referred to as the child class or derived class.

Think of it like a family tree. The superclass is the parent, passing down its traits to its children, the subclasses. The syntax for inheritance is pretty straightforward:

```
class Superclass:
    # Superclass attributes and methods

class Subclass(Superclass):
    # Subclass attributes and methods
```

Let's make this more concrete with a simple example:

```
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        print("The animal makes a sound")

class Dog(Animal):
    def speak(self):
        print(f"{self.name} says Woof!")

# Creating instances
generic_animal = Animal("Generic")
my_dog = Dog("Buddy")

# Using the objects
generic_animal.speak() # Output: The animal makes a sound
my_dog.speak()         # Output: Buddy says Woof!
```

In this example, **Animal** is our superclass, and **Dog** is our subclass. Notice how **Dog** inherits the name attribute from **Animal**, but we've given it a more specific speak method.

This is incredibly useful in real-world programming. For instance, if you're building a game with different character types, you could have a base Character class with common attributes like health and position, then create subclasses like Warrior, Mage, and Archer with their unique abilities.

Inheritance allows subclasses to inherit all methods and properties from their superclass, unless specified otherwise and subclasses can override inherited methods to provide specific implementations.

Python also supports multiple inheritance, meaning a class can inherit from more than one superclass. However, this can get complex, so it's best to keep things simple when you're starting out.

Remember, inheritance is all about reusing code and creating logical hierarchies in your programs. It's a powerful tool, but like any tool, it's important to use it wisely. Don't create complex inheritance structures just for the sake of it - use inheritance when it makes your code clearer and more efficient.

Types of Inheritance

There are five main types of inheritance in Python.

■ Single inheritance is the simplest form of inheritance, where a class inherits from just one parent class.

Here's a simple example:

```
class Animal:
    def speak(self):
        print("I'm an animal")

class Dog(Animal):
    def bark(self):
        print("Woof!")

my_dog = Dog()
my_dog.speak() # Output: I'm an animal
my_dog.bark()  # Output: Woof!
```

In this example, the Dog class inherits from the Animal class, gaining its speak method while also having its own bark method.

■ Multiple inheritance allows a class to inherit from more than one parent class. This can be powerful but also complex if not used carefully.

```
class Flyer:
    def fly(self):
        print("I can fly!")

class Swimmer:
    def swim(self):
        print("I can swim!")

class Duck(Flyer, Swimmer):
    pass

donald = Duck()
donald.fly() # Output: I can fly!
donald.swim() # Output: I can swim!
```

In this case, the Duck class inherits from both Flyer and Swimmer, gaining abilities from both parent classes.

■ Multilevel inheritance involves creating a child class from another child class. It creates a parent-child-grandchild relationship.

```
class Animal:
    def breathe(self):
        print("I can breathe")

class Mammal(Animal):
    def feed_young(self):
        print("I can feed my young")

class Dog(Mammal):
    def bark(self):
        print("Woof!")

my_dog = Dog()
my_dog.breathe() # Output: I can breathe
my_dog.feed_young() # Output: I can feed my young
my_dog.bark() # Output: Woof!
```

Here, Dog inherits from Mammal, which in turn inherits from Animal, creating a chain of inheritance.

■ In hierarchical inheritance, multiple child classes inherit from a single parent class.

```
class Vehicle:
    def move(self):
        print("I can move")

class Car(Vehicle):
    def drive(self):
        print("I can drive")

class Boat(Vehicle):
    def sail(self):
        print("I can sail")

my_car = Car()
my_boat = Boat()
my_car.move() # Output: I can move
my_car.drive() # Output: I can drive
my_boat.move() # Output: I can move
my_boat.sail() # Output: I can sail
```

Both Car and Boat inherit from Vehicle, creating a hierarchy.

■ Hybrid inheritance is a combination of two or more types of inheritance. It's the most complex form and should be used with care.

```
class A:
    def method_a(self):
        print("Method from A")

class B(A):
    def method_b(self):
        print("Method from B")

class C(A):
    def method_c(self):
        print("Method from C")

class D(B, C):
    pass

d = D()
d.method_a() # Output: Method from A
d.method_b() # Output: Method from B
d.method_c() # Output: Method from C
```

In this example, we have a mix of multiple and hierarchical inheritance.

Remember, the goal of inheritance is to promote code reuse and create logical relationships between classes. Always strive for clarity and simplicity in your designs.

Method Inheritance

Method inheritance is a powerful feature in Python that allows us to reuse code and build upon existing functionality. It's like getting a head start on your work by using what's already been created!

When we create a new class based on an existing one, the new child class automatically gets all the methods

from the parent class. This is super helpful because we don't have to rewrite code that already works!

```
class Animal:
    def speak(self):
        print("The animal makes a sound")

class Dog(Animal):
    pass # This means we're not adding anything new
        # right now

# Creating a dog
my_dog = Dog()
my_dog.speak() # Output: The animal makes a sound
```

In this example, the Dog class inherits the speak method from the Animal class. Even though we didn't define speak in the Dog class, we can still use it!

Sometimes, we want our child class to do something differently than the parent class. This is where method overriding comes in handy. To do this, we define a method in the child class with the same name as a method in the parent class. This new method replaces the inherited one.

```
class Animal:
    def speak(self):
        print("The animal makes a sound")

class Dog(Animal):
    def speak(self):
        print("The dog says Woof!")

my_dog = Dog()
my_dog.speak() # Output: The dog says Woof!
```

Sometimes, we want to use the parent class's method and then add some extra functionality. That's where the `super()` function comes in handy!

```
class Animal:
    def speak(self):
        print("The animal makes a sound")

class Dog(Animal):
    def speak(self):
        super().speak() # This calls the parent class's
                        # speak method
        print("The dog also says Woof!")

my_dog = Dog()
my_dog.speak()
# Output:
# The animal makes a sound
# The dog also says Woof!
```

The `super()` function is useful in this scenario. It allows you to call a method from the parent class inside a method of the child class.

For example, here, the `Dog` class inherits from the `Animal` class. The `Dog` class's `speak` method first uses `super().speak()` to call the `speak` method from the `Animal` class, which prints "The animal makes a sound." Then, it adds its own functionality by printing

"The dog also says Woof!" This way, the `Dog` class can extend or modify the behaviour of the `Animal` class while still retaining the base functionality.

Constructor Inheritance

As we have seen, a constructor is a special method called `__init__` that's used to initialize an object when it's created. When we're working with inheritance, understanding how constructors work is crucial.

In inheritance, when we create an object of a child class, Python first calls the constructor of the parent class before executing the child class's constructor.

```
class Animal:
    def __init__(self, name):
        print("Animal constructor called")
        self.name = name

class Dog(Animal):
    def __init__(self, name, breed):
        print("Dog constructor called")
        super().__init__(name) # Calling the base class
        constructor
        self.breed = breed

# Creating a Dog object
my_dog = Dog("Buddy", "Golden Retriever")
```

When we run this code, we'll see:

```
Dog constructor called
Animal constructor called
```

Notice how we used `super().__init__(name)` in the `Dog` class. This is how we call the base class constructor from the derived class. It's a good practice to always call the base class constructor to ensure proper initialization.

Abstract Classes

Abstract Classes will help you design better, more flexible code. Let's break it down step by step!

Imagine you're designing a video game with different types of characters. You know all characters will have some common behaviours, but you don't want to create a generic "Character" that can be instantiated directly. This is where abstract classes come in handy!

An abstract class is like a blueprint for other classes. It can declare methods that must be implemented by its subclasses, but it can't be instantiated itself. In Python, we use the `abc` (Abstract Base Class) module to create abstract classes.

Here's a simple example:

```
from abc import ABC, abstractmethod

class Character(ABC):
    @abstractmethod
    def attack(self):
        pass

    @abstractmethod
    def defend(self):
        pass

class Warrior(Character):
    def attack(self):
        print("Warrior attacks with a sword!")

    def defend(self):
        print("Warrior defends with a shield!")

# This works:
warrior = Warrior()
warrior.attack() # Output: Warrior attacks with a
sword!

# This would raise an error:
# character = Character() # TypeError: Can't
instantiate abstract class Character
```

In this example, `Character` is an abstract class with two abstract methods: `attack` and `defend`. Any class that inherits from `Character` must implement these methods, or it will also be considered abstract.

Abstract classes are great for defining a common interface for a group of related subclasses and for ensuring that certain methods are implemented in derived classes. They provide a structure for class hierarchies.

Composition vs Inheritance

Now, let's talk about two different ways to structure your code: composition and inheritance. You've already seen inheritance in action, but let's compare it with composition.

Inheritance is when a class (subclass) is based on another class (superclass), inheriting its attributes and methods. Composition, on the other hand, is when a class contains instances of other classes as its attributes.

Here's a quick example to illustrate both:

```
# Inheritance
class Animal:
    def make_sound(self):
        pass

class Dog(Animal):
    def make_sound(self):
        return "Woof!"

# Composition
class Engine:
    def start(self):
        return "Engine started"
```

```
class Car:
    def __init__(self):
        self.engine = Engine()

    def start_car(self):
        return self.engine.start()
```

In the inheritance example, `Dog` is a type of `Animal`. In the composition example, a `Car` has an `Engine`, but it's not a type of `Engine`.

The principle "Favor composition over inheritance" suggests that it's often better to use composition when designing your classes. Why? Because:

- ❑ Composition is more flexible. You can change the components of a class more easily than changing its inheritance hierarchy.
- ❑ It reduces coupling between classes, making your code more modular and easier to maintain.
- ❑ It allows for more dynamic behaviour at runtime.

However, inheritance is still useful when you have a clear "is-a" relationship between classes and want to share code.

In practice, you'll often use a combination of both inheritance and composition. The key is to use each where it makes the most sense for your specific situation.

These methods and techniques are all tools in your programming toolbox. As you gain more experience, you'll develop a better intuition for when to use each approach. Don't be afraid to experiment and refactor your code as you learn!

Polymorphism Basics

Polymorphism is a crucial concept in object-oriented programming that'll make your code more flexible and powerful.

Imagine you have a magic box that can hold different types of animals. When you say "speak" to the box, each animal makes its own unique sound. That's essentially what polymorphism is all about!

In programming terms, polymorphism is the ability of objects of different types to respond to the same method call, each in its own way. It's like having a universal remote that works with different brands of TVs, each responding according to its own programming.

```
class Dog:
    def speak(self):
        return "Woof!"

class Cat:
    def speak(self):
        return "Meow!"

class Duck:
    def speak(self):
        return "Quack!"

def animal_sound(animal):
    print(animal.speak())

# Creating objects
dog = Dog()
cat = Cat()
duck = Duck()

# Using polymorphism
animal_sound(dog) # Output: Woof!
animal_sound(cat) # Output: Meow!
animal_sound(duck) # Output: Quack!
```

In this example, `animal_sound()` function doesn't need to know what type of animal it's dealing with. It just calls the `speak()` method, and each animal responds appropriately. That's the beauty of polymorphism!

Method Overriding (in-depth)

Method overriding is when a child class provides a specific implementation for a method that's already defined in its parent class. It's like saying, "Thanks for that method, Mom and Dad, but I've got my own way of doing things!"

Rules for method overriding:

- ❑ The method in the child class must have the same name as the method in the parent class.
- ❑ The method must have the same number of parameters.
- ❑ The method should return the same type of result (though Python is flexible about this).

Let's see an example:

```
class Animal:
    def speak(self):
        return "Some sound"

class Dog(Animal):
    def speak(self):
        return "Woof!"

class Cat(Animal):
    def speak(self):
        return "Meow!"

dog = Dog()
cat = Cat()

print(dog.speak()) # Output: Woof!
print(cat.speak()) # Output: Meow!
```

In this example, both `Dog` and `Cat` override the `speak` method from the `Animal` class.

Mixins and Traits

Mixins are a way to add functionality to a class without using traditional inheritance. They're like little packets of methods that you can "mix in" to your classes. Using mixins to achieve multiple inheritance-like behavior:

```
class SwimMixin:
    def swim(self):
        return "I'm swimming!"

class FlyMixin:
    def fly(self):
        return "I'm flying!"

class Duck(SwimMixin, FlyMixin):
    pass

donald = Duck()
print(donald.swim()) # Output: I'm swimming!
print(donald.fly()) # Output: I'm flying!
```

Here, `Duck` gets capabilities from both `SwimMixin` and `FlyMixin` without traditional inheritance.

Python uses the Method Resolution Order (MRO) to decide which method to call if there are conflicts. The rule of thumb is: "rightmost wins". So, if multiple mixins define the same method, the one from the rightmost mixin in the class definition is used.

Mixins are great for adding optional features to classes without creating complex inheritance hierarchies. But be careful not to overuse mixins, as they can make your code harder to understand if used excessively.

The Template Method Pattern

Imagine you're baking different types of cakes. While each cake is unique, they all follow a similar process: mix ingredients, pour into a pan, bake, and decorate. This is exactly what the Template Method Pattern does in programming!

The Template Method Pattern uses inheritance to define the skeleton of an algorithm in a base class, allowing subclasses to override specific steps without changing the overall structure.

Let's look at a simple example:

```
class Cake:
    def bake(self):
        self.prepare()
        self.mix()
        self.pour()
        self.cook()
```

```
        self.decorate()

    def prepare(self):
        print("Gathering ingredients")

    def mix(self):
        print("Mixing ingredients")

    def pour(self):
        print("Pouring into pan")

    def cook(self):
        print("Baking for 30 minutes")

    def decorate(self):
        print("Adding frosting")

class ChocolateCake(Cake):
    def mix(self):
        print("Mixing ingredients with cocoa")

    def decorate(self):
        print("Adding chocolate chips")

# Usage
vanilla_cake = Cake()
chocolate_cake = ChocolateCake()

print("Baking a vanilla cake:")
vanilla_cake.bake()

print("\nBaking a chocolate cake:")
chocolate_cake.bake()
```

In this example, `Cake` is our base class that defines the template method `bake()`. The `ChocolateCake` class inherits from `Cake` and overrides specific methods to customize the process.

The Template Method Pattern is great for frameworks where you want to define a standard process but allow for customization in specific steps.

The Strategy Pattern

Now, let's say you're building a navigation app. Users might want different types of routes: fastest, scenic, or avoiding tolls. This is where the Strategy Pattern rocks! It uses polymorphism to select different algorithms at runtime. It allows you to define a family of algorithms, encapsulate each one, and make them interchangeable.

Here's a simple example:

```
from abc import ABC, abstractmethod

class RouteStrategy(ABC):
    @abstractmethod
    def calculate_route(self, start, end):
        pass

class FastestRoute(RouteStrategy):
    def calculate_route(self, start, end):
        return f"Calculating fastest route from {start} to {end}"

class ScenicRoute(RouteStrategy):
    def calculate_route(self, start, end):
```

```

        return f"Calculating scenic route from {start}
to {end}"

class NavigationApp:
    def __init__(self, strategy):
        self.strategy = strategy

    def navigate(self, start, end):
        return self.strategy.calculate_route(start, end)

app = NavigationApp(FastestRoute())
print(app.navigate("Home", "Work"))

app.strategy = ScenicRoute()
print(app.navigate("Home", "Beach"))

```

In this example, we can easily switch between different routing strategies without changing the `NavigationApp` class.

Common Pitfalls and Best Practices

Alright, let's finish with some common pitfalls and best practices when it comes to inheritance and polymorphism in Python.

First up, let's talk about deep inheritance hierarchies. Imagine you're building a family tree in code. It might be tempting to create a long chain of classes, like this:

```

class Animal:
    pass

class Mammal(Animal):
    pass

class Carnivore(Mammal):
    pass

class Feline(Carnivore):
    pass

class Lion(Feline):
    pass

```

While this might seem logical, it can lead to some problems:

- It's harder to understand and maintain.
- Changes at the top can have unexpected effects way down the chain.
- It can lead to the "diamond problem" in multiple inheritance.

Try to keep your inheritance hierarchies shallow, usually no more than 2-3 levels deep. Instead of deep hierarchies, consider using composition (where one class contains instances of other classes) or interfaces.

When you're creating a class, it's important to think about whether you want other classes to inherit from it. If you do, you need to design it carefully:

```

class Vehicle:
    def __init__(self, make, model):
        self.make = make
        self.model = model

    def start_engine(self):
        print("Engine started")

class Car(Vehicle):
    def __init__(self, make, model, num_doors):
        super().__init__(make, model)
        self.num_doors = num_doors

```

Here, `Vehicle` is designed to be inherited from. It has a clear, simple interface that `Car` can build upon.

On the flip side, if you don't want a class to be inherited from, you can use naming conventions to indicate this:

```

class _InternalUseOnly:
    # This class is not meant to be inherited from
    pass

```

In Python, leading underscores in class names are a convention indicating that a class is meant for internal use only.

In conclusion, inheritance and polymorphism are powerful tools, but they need to be used wisely. By following these best practices, you'll create code that's easier to understand, maintain, and extend.



NumPy

Imagine you're a chef, and Python is your kitchen. NumPy? Well, that's like getting a whole set of professional-grade tools that make your cooking faster, more efficient, and capable of handling much larger recipes. That's NumPy in a nutshell!

NumPy, which stands for Numerical Python, is a fundamental package for scientific computing in Python. Think of it as the Swiss Army knife for handling large, multi-dimensional arrays and matrices.

In the world of data science and scientific computing, NumPy is like the foundation of a house. Many other important libraries, like Pandas, SciPy, and Matplotlib, are built on top of NumPy. It's essential because it provides:

- ▣ Efficient storage of large arrays of numerical data
- ▣ A vast collection of mathematical functions to operate on these arrays
- ▣ Tools for reading/writing array-based datasets to disk
- ▣ Linear algebra operations, Fourier transforms, and random number generation capabilities

Now, you might be thinking, "Can't I just use Python lists for all this?" Well, you could, but that would be like trying to cut down a forest with a pocket knife when you have a chainsaw available!

NumPy arrays are much faster than Python lists for numerical operations - about 10-100 times faster in many cases! NumPy also uses less memory to store data compared to Python lists. It provides a wide range of mathematical operations that work directly on arrays, which aren't available for standard Python lists. Finally, with NumPy, you can easily perform operations on entire arrays without explicit loops, a concept called vectorization.

NumPy is a vast library with many capabilities. We're not going to cover everything but we'll lay a foundation that will serve you well as you dive deeper into the world of scientific computing and data analysis with Python!

Installing and Importing NumPy

Installing NumPy is very straightforward. You can use the **pip** command (or **pip3** if you are using a Mac or Linux) in your terminal to do so:

```
pip install numpy
```

Once NumPy has installed you can import it into your programs with

```
import numpy as np
```

It is a general convention to **import numpy as np** and you will see it done like this in most programs.

NumPy arrays

At the heart of NumPy are its arrays. A NumPy array is a grid of values, all of the same type, indexed by a tuple of non-negative integers. In simpler terms, you can think of it as a container that can hold a bunch of numbers, all neatly organized in a specific shape.

While NumPy arrays might seem similar to Python lists at first glance, they have some key differences:

- ▣ All elements in a NumPy array must be of the same data type. This isn't the case with Python lists, which can contain elements of different types.
- ▣ Once created, NumPy arrays have a fixed size. You can't add or remove elements from them like you can with Python lists.
- ▣ NumPy arrays support vectorized operations, which means you can perform operations on entire arrays without explicit loops.

Creating NumPy arrays

Let's start with the basics. You can create a NumPy array from a regular Python list using **np.array()**.

```
import numpy as np

# Creating a 1D array of latitudes
lat_list = [40.7128, 34.0522, 41.8781, 29.7604]
lat_array = np.array(lat_list)
print(lat_array)
# Output: [40.7128 34.0522 41.8781 29.7604]

# Creating a 2D array of lat-long pairs
coord_list = [[40.7128, -74.0060], [34.0522, -118.2437],
              [41.8781, -87.6298]]
coord_array = np.array(coord_list)
print(coord_array)

# Output:
# [[ 40.7128 -74.006 ]
#  [ 34.0522 -118.2437]
#  [ 41.8781 -87.6298]]
```

However, NumPy also provides methods to create NumPy arrays that are filled with values, which you can later change if needed.

For example, **np.zeros()** creates an array filled with zeros. This is a great method for initializing data structures or creating masks.

```
# Create a 3x3 array of zeros (could represent an empty
# elevation grid)
elevation_grid = np.zeros((3, 3))
print(elevation_grid)

# Output:
# [[0. 0. 0.]
#  [0. 0. 0.]
#  [0. 0. 0.]
```

Inside the `np.zeros()` function we use a tuple to indicate the shape and size of the array. In this case we are creating a two-dimensional array with three values on each axis.

Similar to `zeros()`, `ones()` fills the array with, you guessed it, ones. This is particularly useful for creating base layers or multipliers. Just as with `zeros()`, `ones()` takes a tuple as a parameter to initialise the shape and size of the array.

```
# Create a 1D array of ones (could represent a uniform
weight for each data point)
weights = np.ones(5)
print(weights)

# Output: [1. 1. 1. 1. 1.]
```

We can create an array without initializing the entries by using `np.empty()`. This is faster than `zeros()` or `ones()` but the initial values are unpredictable. When using `np.empty()`, Python will take a random section of memory and fill the array with its values. That's why you should always initialise `np.empty()` array before using them to avoid unpredictable behaviour.

```
# Create an empty 2x2 array
quick_array = np.empty((2, 2))
print(quick_array)

# Output might look like:
# [[4.9e-324 9.9e-324]
#  [1.5e-323 2.0e-323]]
```

`np.arange(start, end, step)` creates an array with a sequence of numbers. Think of it as Python's `range()` function on steroids. `np.arange()` takes three parameters, a starting value, an end value (not included), and a step size.

```
# Create an array of years from 2000 to 2009
years = np.arange(2000, 2010)
print(years)

# Output: [2000 2001 2002 2003 2004 2005 2006 2007 2008
2009]
# Create an array of temperatures with 0.5°C increments
temps = np.arange(20, 25, 0.5)
print(temps)

# Output: [20. 20.5 21. 21.5 22. 22.5 23. 23.5 24.
24.5]
```

If we know the range that we want to create values over but want to set a predefined number of values, we can use the `np.linspace(start, end, amount)` function. It creates an array with a specified number of elements, spaced evenly between a start and end value.

```
# Create 5 evenly spaced elevation levels between 0 and
1000 meters
elevation_levels = np.linspace(0, 1000, 5)
print(elevation_levels)

# Output: [ 0. 250. 500. 750. 1000.]
```

NumPy also allows you to create arrays with random numbers drawn from specified distributions.

`np.random.rand(dimensions*)` generates an array of random numbers between 0 and 1, drawn from a uniform distribution. The `dimensions` parameter indicates the length of each axis. For example, `np.random.rand(3, 2)` will generate a 3 x 2 array of random number from a uniform distribution.

```
# Generate 5 random precipitation values (0-1 scale)
precipitation = np.random.rand(5)
print(precipitation)

# Output might look like: [0.23651224 0.41900661
0.84902712 0.29261896 0.13908253]
```

`np.random.randn(dimensions*)` generates an array of random numbers drawn from a standard normal distribution (mean 0, standard deviation 1).

```
# Generate 5 random temperature anomalies (in °C)
temp_anomalies = np.random.randn(5)
print(temp_anomalies)

# Output might look like: [-0.84153612 0.23456971
1.2048685 -0.38405435 0.13376307]
```

`np.random.randint(low, high, size)` generates an array of random integers within a specified range. The `high` parameter, indicating the upper limit of the range is excluded from the range.

```
# Generate 10 random elevation values between 0 and 1000
meters
random_elevations = np.random.randint(0, 1001, 10)
print(random_elevations)
# Output might look like: [245 31 892 421 760 582 102
494 795 430]
```

Let's simulate a simple elevation profile for a hiking trail:

```
# Generate a 50-point elevation profile
trail_length = 50
base_elevation = 1000
elevation_profile = base_elevation +
np.cumsum(np.random.randint(-10, 11, trail_length))

print("Elevation profile of the trail:")
print(elevation_profile)

# Calculate some statistics
print(f"Starting elevation: {elevation_profile[0]} m")
print(f"Ending elevation: {elevation_profile[-1]} m")
print(f"Highest point: {np.max(elevation_profile)} m")
print(f"Lowest point: {np.min(elevation_profile)} m")
```

This script creates a random elevation profile, simulating ups and downs along a trail, and then calculates some basic statistics about the hike.

Appending and Adding Elements

In Python lists, you might be familiar with the `append()` method to add elements. NumPy arrays, however, are of fixed size once created, meaning you cannot directly append elements to them like you can with lists. Instead, NumPy provides specific functions to create a new array that includes the additional elements.

Using `np.append()`

The `np.append()` function is used to append elements to an array. This function does not modify the original array but returns a new array with the added elements.

```
import numpy as np

# Creating an initial array
arr = np.array([1, 2, 3, 4])

# Appending a single element
arr = np.append(arr, 5)
print("Array after appending an element:", arr)
# Output: [1 2 3 4 5]

# Appending multiple elements
new_arr_multi = np.append(arr, [6, 7, 8])
print("Array after appending multiple elements:",
      new_arr_multi)
# Output: [1 2 3 4 6 7 8]
```

We assigned the newly created array after the `np.append()` function was called back to the `arr` variable. That way we have essentially added elements to the array.

Inserting elements with `np.insert()`

If you need to insert elements at a specific index in an array, you can use the `np.insert()` function. Like `np.append()`, this function returns a new array.

```
# Inserting an element at a specific position
arr_with_insert = np.insert(arr, 2, 99)
print("Array after inserting an element at index 2:",
      arr_with_insert)
# Output: [ 1  2 99  3  4]
```

Using `np.add()` to add elements

While `np.append()` is used to add new elements to the end of an array, `np.add()` allows you to perform element-wise addition between two arrays, effectively

adding elements together rather than extending the array's size.

```
import numpy as np

# Creating two arrays
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])

# Adding the two arrays element-wise
result = np.add(arr1, arr2)
print("Result of np.add:", result)

# Output: [5 7 9]
```

In this example, `np.add()` adds the elements of `arr1` and `arr2` together, element by element. It's important to note that `np.add()` does not change the size of the array but rather combines corresponding elements from two arrays.

Considerations When Adding Elements

While adding elements to arrays is straightforward with these functions, it's important to note that these operations often involve creating a new array, which can be computationally expensive for large arrays. Therefore, it is generally better to preallocate arrays with the necessary size if the number of elements is known in advance.

Array attributes

When working with geospatial data, NumPy becomes your best friend (after the print function, of course), helping you manipulate and analyse large datasets efficiently. Let's look at some key attributes of NumPy arrays that you'll use all the time:

■ **shape:**

The `shape` attribute tells us the dimensions of our array. It's like asking, "How big is this data structure?" Let's say we have an array representing elevation data for a 5x5 grid:

```
import numpy as np

elevation = np.array([[100, 120, 130, 110, 105],
                      [95, 115, 140, 130, 120],
                      [90, 110, 135, 140, 130],
                      [85, 100, 120, 135, 125],
                      [80, 95, 110, 120, 115]])

print(elevation.shape)
# Output: (5, 5)
```

This tells us we have a 5x5 grid. The shape is always returned as a tuple, even for 1D arrays.

■ `ndim`:

The `ndim` attribute gives us the number of dimensions in our array. It's like asking, "How many directions can I move in this data?" Using our elevation example:

```
print(elevation.ndim)
# Output: 2
```

This tells us we're working with a 2-dimensional array. If we had a time series of elevation data, it might be 3-dimensional!

■ `size`:

The `size` attribute tells us the total number of elements in the array. This is handy when you need to know how much data you're dealing with, regardless of its shape.

```
print(elevation.size)
# Output: 25
```

■ `dtype`:

The `dtype` attribute tells us the data type of the elements in the array.

```
print(elevation.dtype)
# Output: int64 (or similar, depending on your system)
```

This tells us we're working with 64-bit integers.

Let's say you're working with a large dataset of GPS coordinates. You might create an array like this:

```
gps_data = np.array([[40.7128, -74.0060], # New York
                    [51.5074, -0.1278], # London
                    [-33.8688, 151.2093], # Sydney
                    [35.6762, 139.6503]]) # Tokyo

print("Shape:", gps_data.shape)
print("Dimensions:", gps_data.ndim)
print("Size:", gps_data.size)
print("Data type:", gps_data.dtype)

# Output:
# Shape: (4, 2)
# Dimensions: 2
# Size: 8
# Data type: float64
```

This tells us we have 4 locations, each with 2 coordinates (latitude and longitude), for a total of 8 data points, stored as 64-bit floating-point numbers for high precision.

Practical tips:

- Use these attributes to quickly understand your data's structure without printing the entire array.
- The `shape` attribute is particularly useful for reshaping arrays or ensuring compatibility in operations.
- Pay attention to `dtype` when performing calculations – mixing types can lead to unexpected results or loss of precision.
- When working with very large datasets, consider using smaller dtypes (like `float32` instead of `float64`) to save memory if the precision is sufficient for your needs.

Array indexing and slicing

One of the most important skills in using NumPy is understanding how to access and manipulate specific elements or sections of these arrays. This is where indexing and slicing come into play.

Imagine you have a large dataset of geographical coordinates. You might want to access a single point, a range of points, or select points based on certain conditions. NumPy provides various methods to do this efficiently, which we'll explore now.

Single Element Access

Let's start with the basics – accessing a single element in a NumPy array.

```
import numpy as np

# Create a 2D array of latitude and longitude coordinates
coordinates = np.array([[40.7128, -74.0060], # New York
                       [51.5074, -0.1278], # London
                       [35.6762, 139.6503], # Tokyo
                       [-33.8688, 151.2093]]) # Sydney

print(coordinates)

# Access the latitude of New York (first row, first column)
ny_lat = coordinates[0, 0]
print(f"Latitude of New York: {ny_lat}")

# Access the longitude of Tokyo (third row, second column)
tokyo_lon = coordinates[2, 1]
print(f"Longitude of Tokyo: {tokyo_lon}")
```

In this example, we use square brackets `[]` to access individual elements. The first number is the row index, and the second is the column index. Remember, NumPy uses zero-based indexing, so the first element is at index 0.

Slicing

Slicing allows us to extract a portion of the array, just like with lists and strings. The basic syntax is `array[start:stop:step]`.

```
# Basic slicing: Get the latitudes of all cities
all_latitudes = coordinates[:, 0]
print("All latitudes:", all_latitudes)

# Get the coordinates of the first three cities
first_three_cities = coordinates[:3]
print("First three cities:", first_three_cities)

# Advanced slicing: Get every second city, both lat and lon
every_second_city = coordinates[::2]
print("Every second city:", every_second_city)
```

If you omit a number in the slice, Python assumes you want to go to the beginning (if omitting the start), to the end (if omitting the stop), or use a step of 1 (if omitting the step).

Boolean Indexing

Boolean indexing is a powerful feature that allows us to select elements based on conditions.

```
# Select all points with latitude > 40
northern_cities = coordinates[coordinates[:, 0] > 40]
print("Cities in the Northern Hemisphere:", northern_cities)

# Select all points in the Western Hemisphere (longitude < 0)
western_cities = coordinates[coordinates[:, 1] < 0]
print("Cities in the Western Hemisphere:", western_cities)
```

This is incredibly useful when you want to filter your data based on certain criteria. It's like asking NumPy, "Give me all the points that satisfy this condition."

Fancy Indexing

Fancy indexing (great name!) allows us to select elements using arrays of indices.

```
# Create an array of indices
indices = np.array([0, 2]) # We want the 1st and 3rd cities

# Use fancy indexing to get these cities
selected_cities = coordinates[indices]
print("Selected cities (New York and Tokyo):", selected_cities)

# We can also use a list of indices
selected_columns = coordinates[:, [1, 0]] # Swap longitude and latitude
print("Coordinates with swapped columns:", selected_columns)
```

Fancy indexing is helpful when you need to rearrange your data or select specific, non-contiguous elements.

Let's say you're working on a project to analyse climate data for coastal cities. You have an array of temperatures corresponding to your coordinates:

```
temperatures = np.array([20, 15, 25, 18]) # Corresponding to our cities

# Select temperatures of cities with longitude > 0 (Eastern Hemisphere)
eastern_temps = temperatures[coordinates[:, 1] > 0]
print("Temperatures of Eastern Hemisphere cities:", eastern_temps)

# Calculate the average temperature of these cities
avg_eastern_temp = np.mean(eastern_temps)
print(f"Average temperature in the Eastern Hemisphere: {avg_eastern_temp:.2f}")
```

This demonstrates how you can combine boolean indexing with data analysis to extract meaningful insights from your geospatial data.

Array reshaping

One of NumPy's most useful features is its ability to manipulate arrays efficiently. There are many methods for doing this.

Reshaping Arrays with `reshape()` Function

This nifty little tool allows you to change the shape of an array without changing its data. It's like rearranging the furniture in your room - same stuff, different layout. All you need to do is give it the length of your new axes as parameters.

Here's how it works:

```
import numpy as np

# Let's create a simple array of numbers from 1 to 12
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
print("Original array:", arr)

# Now, let's reshape it into a 3x4 array
reshaped_arr = arr.reshape(3, 4)
print("Reshaped array:")
print(reshaped_arr)
```

When you run this, you'll see:

```
Original array: [ 1  2  3  4  5  6  7  8  9 10 11 12]
Reshaped array:
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

Cool, right? We've taken our flat list and turned it into a 3x4 grid. The total number of elements must remain

the same. You can't reshape a 12-element array into a 3x5 array, which would be 15 elements!

You can also use -1 as one of the dimensions, and NumPy will figure out what it should be:

```
# Reshape into 2 rows, let NumPy figure out the columns
auto_resaped = arr.reshape(2, -1)
print("Auto-resaped array:")
print(auto_resaped)
```

This will give you:

```
Auto-resaped array:
[[ 1  2  3  4  5  6]
 [ 7  8  9 10 11 12]]
```

NumPy calculated that if we want 2 rows, we need 6 columns to fit all 12 elements.

Flattening Arrays

Flattening array is essentially the opposite of reshaping - you're taking a multi-dimensional array and turning it into a 1D array. There are a couple of ways to do this:

■ Using reshape(-1):

```
# Let's start with our 3x4 array from before
multi_dim = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9,
10, 11, 12]])
print("Original multi-dimensional array:")
print(multi_dim)

# Flatten using reshape(-1)
flattened = multi_dim.reshape(-1)
print("Flattened array:")
print(flattened)
```

■ Using flatten() method:

```
# Flatten using flatten() method
flattened2 = multi_dim.flatten()
print("Flattened array using flatten():")
print(flattened2)
```

Both of these will give you the same result:

```
Original multi-dimensional array:
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
Flattened array:
[ 1  2  3  4  5  6  7  8  9 10 11 12]
Flattened array using flatten():
[ 1  2  3  4  5  6  7  8  9 10 11 12]
```

Useful Tips:

- Always check your array's shape before and after reshaping to make sure you got what you wanted.

- Remember, reshaping doesn't change the data, just how it's arranged. The total number of elements always stays the same.
- When in doubt, use -1 in your reshape command to let NumPy figure out the appropriate dimension.
- Flattening can be really useful for vectorizing operations or preparing data for certain machine learning algorithms.

Array concatenation and splitting

Being able to join arrays together and break them apart are super handy skills when you're working with data, like combining different map layers or splitting up large datasets. So, let's get started!

Array Concatenation

Imagine you have two separate datasets of data for neighbouring regions. Wouldn't it be great to combine them into one big dataset? That's where array concatenation comes in!

■ np.concatenate()

This is your go-to function for joining arrays:

```
import numpy as np

mountain_range_1 = np.array([[100, 200, 300], [400, 500, 600]])
mountain_range_2 = np.array([[700, 800, 900], [1000, 1100,
1200]])

# Combine them vertically (stacking rows)
combined_ranges = np.concatenate((mountain_range_1,
mountain_range_2))
print("Combined mountain ranges:")
print(combined_ranges)
```

This will stack the two mountain ranges on top of each other. Cool, right?

■ np.vstack(), np.hstack(), and np.dstack()

These functions are like specialized tools for vertical, horizontal, or dimensional stacking. They let you decide whether you want the data on top of each other or next to each other once joined or joined along another axis.

```
# Vertical stacking (same as before)
vertical_stack = np.vstack((mountain_range_1,
mountain_range_2))
print("Vertical stack:")
print(vertical_stack)

# Horizontal stacking (side by side)
horizontal_stack = np.hstack((mountain_range_1,
mountain_range_2))
print("Horizontal stack:")
print(horizontal_stack)
```

Use `vstack()` when you want to add more rows, and `hstack()` when you want to add more columns. Use `dstack()` if you want to stack along another axis. Think of it as stacking plates vertically or lining up books horizontally on a shelf.

Array Splitting

Now, let's say you have a large satellite image and you want to break it into smaller chunks for processing. That's where array splitting comes in handy!

■ `np.split(array, num_parts)`

This function splits an array into multiple sub-arrays. It takes two parameters: the array that you'd like to split and the number of parts that you'd like to split it up into.

```
# Create a large array representing a satellite image
satellite_image = np.arange(16).reshape(4, 4)
print("Original satellite image:")
print(satellite_image)

# Split it into two equal parts vertically
split_image = np.split(satellite_image, 2)
print("Split image:")
for part in split_image:
    print(part)
```

■ `np.hsplit()`, `np.vsplit()`, and `np.dsplit()`

These are specialized versions for horizontal and vertical splitting. They work like `np.split()` except that they operate either on the horizontal or vertical axis.

```
# Horizontal split (splitting columns)
h_split = np.hsplit(satellite_image, 2)
print("Horizontal split:")
for part in h_split:
    print(part)

# Vertical split (splitting rows)
v_split = np.vsplit(satellite_image, 2)
print("Vertical split:")
for part in v_split:
    print(part)
```

Combined, concatenation and splitting provide a powerful toolset. Imagine you're analysing climate data across different regions. You could use concatenation to combine temperature data from multiple weather stations into a single array. Then, you might use splitting to separate this data by season or geographical area for more detailed analysis.

Data types in NumPy

NumPy uses its own data types, which are optimized for performance and memory usage and allow it to

perform large-scale operations effortlessly. Let's have a look at them

Integer Types in NumPy

NumPy provides several integer types with different sizes:

- `int8`: 8-bit integer (-128 to 127)
- `int16`: 16-bit integer (-32,768 to 32,767)
- `int32`: 32-bit integer (-2,147,483,648 to 2,147,483,647)
- `int64`: 64-bit integer (-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807)

These different sizes allow you to optimize memory usage based on your data's range.

Let's see an example using latitude coordinates:

```
import numpy as np

# Creating an array of latitudes
latitudes = np.array([37, 40, 42, 39], dtype=np.int8)
print(latitudes)
print(latitudes.dtype)
```

Output:

```
[ 37  40  42  39]
int8
```

In this example, we used `int8` because our latitude values easily fit within its range. This saves memory compared to using larger integer types and speeds up your execution time.

You should try to always choose the smallest data type that can accommodate your data range to optimize memory usage.

Float Types in NumPy

For decimal numbers, NumPy offers these float types:

- `float16`: Half precision float
- `float32`: Single precision float
- `float64`: Double precision float (default)

```
# Creating an array of longitudes
longitudes = np.array([-122.4194, -74.0060, 0.1276, 151.2093], dtype=np.float32)
print(longitudes)
print(longitudes.dtype)
```

Output:

```
[-122.4194  -74.006    0.1276  151.2093 ]
float32
```

We used float32 here as it provides sufficient precision for geographic coordinates while using less memory than float64.

Boolean Type in NumPy

The boolean type in NumPy is simple – it's the same as the build-in boolean type: True or False. It's great for creating masks or conditional arrays.

Let's see an example where we check if locations are in the Northern Hemisphere:

```
latitudes = np.array([37, -23, 51, -14])
northern_hemisphere = latitudes > 0
print(northern_hemisphere)
print(northern_hemisphere.dtype)
```

Output:

```
[ True False  True False]
bool
```

This boolean array can be used to filter our data easily.

Complex Number Type in NumPy

While less common in applications, NumPy also supports complex numbers. These are useful in signal processing or certain mathematical operations.

```
# Creating a complex number array
complex_data = np.array([1+2j, 3-4j, 5+6j])
print(complex_data)
print(complex_data.dtype)
```

Output:

```
[1.+2.j 3.-4.j 5.+6.j]
complex128
```

String Type in NumPy

NumPy can also handle strings, which is useful for labelling or categorical data. They behave similarly to Python strings except that they are fixed in length.

Here's an example with city names:

```
cities = np.array(['New York', 'London', 'Tokyo',
                  'Sydney'])
print(cities)
print(cities.dtype)
```

Output:

```
['New York' 'London' 'Tokyo' 'Sydney']
<U7
```

The '<U7' means it's using Unicode strings with a maximum length of 7 characters.

Let's combine multiple data types in a structured array for a simple geospatial dataset:

```
data = np.array([('New York', 40.7128, -74.0060, True),
                ('Tokyo', 35.6762, 139.6503, False),
                ('London', 51.5074, -0.1278, True)],
                dtype=[('city', 'U10'), ('lat', 'f4'),
                      ('lon', 'f4'), ('visited', '?')])

print(data)
print(data['city'])
print(data['visited'])
```

Output:

```
[('New York', 40.7128, -74.006 , True)
 ('Tokyo',    35.6762, 139.6503, False)
 ('London',   51.5074, -0.1278,  True)]
['New York' 'Tokyo' 'London']
[ True False  True]
```

This structured array combines strings, floats, and booleans to create a rich dataset. Using the **dtype** parameter, we were able to manually set the data type for each column, a string with max character of 10, two 32-bit floats and an unknown.

Type conversion

One of the key features of NumPy is its ability to handle different data types efficiently, and that's where type conversion comes into play.

Type Conversion with **astype()**:

In NumPy, the **astype()** method is your go-to tool for converting the data type of an array. It lets you transform your data from one type to another, making it incredibly versatile for various data manipulation tasks.

The syntax for **astype()** is straightforward:

```
new_array = original_array.astype(new_type)
```

Here's a simple example:

```
import numpy as np

# Create a NumPy array of integers
elevations = np.array([100, 150, 200, 250])
print("Original array:", elevations)
print("Data type:", elevations.dtype)

# Convert to float
elevations_float = elevations.astype(float)
print("Converted array:", elevations_float)
print("New data type:", elevations_float.dtype)
```

Output:

```
Original array: [100 150 200 250]
Data type: int64
Converted array: [100. 150. 200. 250.]
New data type: float64
```

In this example, we converted an array of integer elevations to floating-point numbers. This could be useful when you need to perform precise calculations with elevation data.

Broadcasting

Broadcasting is NumPy's way of performing operations on arrays of different shapes. It allows NumPy to work with arrays that you might think shouldn't work together. The beauty of broadcasting is that it lets you perform operations without actually copying data, which makes your code more memory-efficient and often faster.

Basic Concept and Rules

- ❏ If the arrays have a different number of dimensions, the shape of the one with fewer dimensions is "padded" with ones on the left side.
- ❏ If the shape of the arrays doesn't match in any dimension, the array with shape equal to 1 in that dimension is stretched to match the other shape.
- ❏ If in any dimension the sizes disagree and neither is equal to 1, an error is raised.

Let's break these down with some examples.

■ Example 1: Adding a scalar to an array

Let's say we have an array of temperatures for different locations, and we want to increase all temperatures by 2 degrees 😊

```
import numpy as np

temperatures = np.array([20, 25, 22, 18, 23])
increase = 2

new_temperatures = temperatures + increase
print(new_temperatures) # Output: [22 27 24 20 25]
```

Here, NumPy is broadcasting the scalar value 2 to match the shape of the temperatures array. It's as if NumPy created a new array [2, 2, 2, 2, 2] behind the scenes to match the dimensions between arrays.

■ Example 2: Adding a 1D array to a 2D array

Imagine we have temperature data for multiple cities over a week, and we want to adjust each day's temperature based on a daily offset:

```
temperatures = np.array([
    [20, 22, 23, 19, 18, 20, 22], # City 1
    [25, 27, 28, 26, 25, 27, 28], # City 2
    [18, 20, 22, 17, 16, 19, 21]  # City 3
])

daily_offset = np.array([0, 1, -1, 2, 0, -2, 1])

adjusted_temperatures = temperatures + daily_offset
print(adjusted_temperatures)
```

Output:

```
[[20 23 22 21 18 18 23]
 [25 28 27 28 25 25 29]
 [18 21 21 19 16 17 22]]
```

In this case, NumPy broadcasts the 1D `daily_offset` array across each row of the 2D `temperatures` array.

■ Example 3: Normalizing 3D data

Let's say you're working with satellite imagery data, and you want to normalize the pixel values across different spectral bands. Each band might have a different mean and standard deviation:

```
# Simulated satellite imagery data (3 bands)
satellite_data = np.random.rand(3, 100, 100) # 3 bands,
100x100 pixels

# Mean and std dev for each band
band_means = np.array([0.5, 0.6, 0.4]).reshape(3, 1, 1)
band_stds = np.array([0.1, 0.15, 0.08]).reshape(3, 1, 1)

# Normalize the data
normalized_data = (satellite_data - band_means) /
band_stds

print(normalized_data.shape) # Output: (3, 100, 100)
```

Here, broadcasting allows us to subtract the mean and divide by the standard deviation for each band without explicitly looping over the bands.

Practical tips:

- ❏ Visualization helps: If you're unsure about how broadcasting will work, try visualizing the shapes of your arrays. NumPy will attempt to match them from the rightmost dimension.
- ❏ Use `np.newaxis` or `None`: You can add new dimensions to your arrays using `np.newaxis` or `None`. This can help you reshape your arrays for broadcasting:

```
a = np.array([1, 2, 3])
b = np.array([[1], [2], [3]])

c = a + b # This will raise an error

# But this will work:
c = a[np.newaxis, :] + b
# or
c = a[None, :] + b
```

- Be cautious with large arrays: while broadcasting can save memory, it might sometimes be more efficient to pre-allocate an array of the correct shape for very large datasets.
- Use broadcasting for elegant code: Broadcasting often allows you to write more concise and readable code compared to explicit looping.

Remember, broadcasting is a powerful tool, but it can also lead to subtle bugs if you're not careful. Always double-check that your arrays are being broadcast in the way you intend.

Basic array operations

Let's dive into some basic array operations that will make your life easier.

Element-wise Operations

Element-wise operations in NumPy allow us to perform calculations on each element of an array individually. This is super handy when you need to apply the same operation to every point in your dataset.

Let's say we have an array representing elevation data in meters, and we want to convert it to feet:

```
import numpy as np

# Create an array of elevation data in meters
elevation_m = np.array([100, 150, 200, 250, 300])

# Convert to feet (1 meter = 3.28084 feet)
elevation_ft = elevation_m * 3.28084

print("Elevation in feet:", elevation_ft)
```

Output:

```
Elevation in feet: [328.084 492.126 656.168 820.21
 984.252]
```

See how easy that was? We multiplied the entire array by a single value, and NumPy took care of applying that operation to each element.

This works with addition, subtraction, and division too. For example, if we wanted to adjust our elevation data by adding 10 meters to each point:

```
adjusted_elevation = elevation_m + 10
print("Adjusted elevation:", adjusted_elevation)
```

Comparison Operations

Comparison operations in NumPy allow us to compare arrays element-wise, which is great for

things like finding areas above or below a certain threshold.

Let's say we want to find all elevations above 200 meters in our data:

```
high_elevation = elevation_m > 200
print("High elevation mask:", high_elevation)
print("High elevation values:",
      elevation_m[high_elevation])
```

Output:

```
High elevation mask: [False False False  True  True]
High elevation values: [250 300]
```

The comparison creates a boolean mask that we can use to index our original array. This is incredibly useful for filtering data based on certain conditions.

Logical Operations

Logical operations allow us to combine multiple conditions, which is perfect for more complex filtering scenarios. Imagine we want to find elevations between 150 and 250 meters:

```
mid_elevation = (elevation_m >= 150) & (elevation_m <=
250)
print("Mid-elevation mask:", mid_elevation)
print("Mid-elevation values:",
      elevation_m[mid_elevation])
```

Output:

```
Mid-elevation mask: [False  True  True  True False]
Mid-elevation values: [150 200 250]
```

Here, we used the `&` (and) operator to combine two conditions. NumPy also supports `|` (or) and `~` (not) for logical operations.

Let's put this all together in a more realistic scenario. Imagine we're analysing a digital elevation model (DEM) to find suitable areas for a wind farm. We want areas above 200m elevation, but below 500m (to avoid extreme conditions):

```
# Simulated DEM data
dem = np.array([
    [150, 180, 210, 240, 270],
    [160, 190, 220, 250, 280],
    [170, 200, 230, 260, 290],
    [180, 210, 240, 270, 300],
    [190, 220, 250, 280, 310]
])

# Find suitable areas
suitable_areas = (dem > 200) & (dem < 500)

print("Suitable areas mask:")
print(suitable_areas)

print("\nElevation of suitable areas:")
```

```
print(dem[suitable_areas])
```

Output:

```
Suitable areas mask:
[[False False  True  True  True]
 [False False  True  True  True]
 [False False  True  True  True]
 [False  True  True  True  True]
 [False  True  True  True  True]]

Elevation of suitable areas:
[210 240 270 220 250 280 230 260 290 210 240 270 300 220
 250 280 310]
```

Universal functions (ufuncs)

Now, imagine you have a huge array of numbers, like elevations across a mountain range, and you need to perform the same mathematical operation on every single one of those numbers. That's where ufuncs, or universal functions, come in handy! They're essentially functions that operate on NumPy arrays element by element, but much faster than if you were to use a regular Python function or a loop.

- ▣ Speed: They're blazing fast, especially with large datasets.
- ▣ Simplicity: They make your code cleaner and easier to read.
- ▣ Versatility: There's a ufunc for almost every mathematical operation you can think of.

Let's look at some common ufuncs and how we might use them with geospatial data:

`np.sqrt()` - Square Root

This ufunc calculates the square root of each element in an array.

```
import numpy as np

# Let's say we have an array of areas (in square
# kilometers)
areas = np.array([100, 225, 400, 625])

# We can calculate the side lengths of square plots with
# these areas
side_lengths = np.sqrt(areas)

print("Areas:", areas)
print("Side lengths:", side_lengths)
```

Output:

```
Areas: [100 225 400 625]
Side lengths: [10. 15. 20. 25.]
```

■ `np.exp()` - Exponential

The exp ufunc calculates e raised to the power of each element in the array.

```
# Let's say we're modelling population growth in
# different regions
growth_rates = np.array([0.01, 0.02, 0.03, 0.04])
time_years = 10

# Calculate population multiplier after 10 years
population_multiplier = np.exp(growth_rates *
time_years)

print("Growth rates:", growth_rates)
print("Population multipliers after 10 years:",
population_multiplier)
```

■ `np.log()` - Natural Logarithm

The log ufunc computes the natural logarithm of each element in the array.

```
# Let's say we're working with earthquake magnitudes
magnitudes = np.array([2, 3, 4, 5, 6])

# The energy released in an earthquake is approximately
# 10^(1.5 * magnitude)
# We can use log to compare energy releases
energy_ratios = np.log(np.power(10, 1.5 * magnitudes))

print("Magnitudes:", magnitudes)
print("Log of energy ratios:", energy_ratios)
```

Output:

```
Magnitudes: [2 3 4 5 6]
Log of energy ratios: [ 6.90775528  9.21034037
11.51292546 13.81551055 16.11809565]
```

You can apply multiple ufuncs in a single line of code. For example: `np.sqrt(np.abs(np.sin(x)))`.

Array input/output

NumPy gives us two main functions for saving our arrays: `np.save()` and `np.savez()`. They each serve a slightly different function but are both responsible for saving your data.

■ `np.save(filename, array)`

This function saves a single array to a file in .npy format – no frills. Here's how you use it:

```
import numpy as np

# Let's create a simple array of latitude coordinates
latitudes = np.array([37.7749, 40.7128, 34.0522,
41.8781, 29.7604])

# Now, let's save it to a file
np.save('latitudes.npy', latitudes)
```

In this example, we've saved our latitudes array to a file called `'latitudes.npy'`. The .npy extension is automatically added if you don't include it.

■ np.savez(filename, arrays*)

Now, `np.savez()`, on the other hand, allows you to save multiple arrays into a single file with a `.npz` extension. Each array is accessible by a key. Here's how it works:

```
# Let's add an array of longitude coordinates
longitudes = np.array([-122.4194, -74.0060, -118.2437, -87.6298, -95.3698])

# Now, let's save both arrays in one file
np.savez('coordinates.npz', lats=latitudes, longs=longitudes)
```

In this case, we've saved both our latitudes and longitudes arrays in a single file called `'coordinates.npz'`. We've given each array a name ('lats' and 'longs') which we'll use to access them later.

Now that we've packed our data, let's learn how to unpack it. For this, we use the `np.load()` function.

■ Loading .npz files: np.load(filename)

To load a single array saved with `np.save()`, it's straightforward:

```
# Load the latitudes array
loaded_latitudes = np.load('latitudes.npy')
print(loaded_latitudes)
```

This will print out the latitudes array we saved earlier.

■ Loading .npz files: np.load(filename)

When loading `.npz` files, we need to specify which array we want to access:

```
# Load the coordinates file
loaded_coordinates = np.load('coordinates.npz')

# Access the latitudes and longitudes
loaded_lats = loaded_coordinates['lats']
loaded_longs = loaded_coordinates['longs']

print("Latitudes:", loaded_lats)
print("Longitudes:", loaded_longs)
```

This will print out both our latitudes and longitudes arrays.

Imagine you're working on a project analysing weather patterns across different cities. You have large arrays of temperature data for each city, collected over several years. Instead of recalculating or re-downloading these data each time you run your analysis, you can save it to a file and load it when needed:

```
temperatures = np.random.rand(5, 365*3) * 30
# Random temps between 0 and 30

# Save the data
```

```
np.save('temperature_data.npy', temperatures)

# Later, when you need to use it again:
loaded_temperatures = np.load('temperature_data.npy')

# Now you can perform your analysis
average_temps = np.mean(loaded_temperatures, axis=1)
print("Average temperatures for each city:",
      average_temps)
```

If you are unsure which keys are contained inside an `.npz` file, you can use the `keys()` method.

```
# Load the data from file
data = np.load('coordinates.npz')
for key in data.keys():
    print(key)

# Output
# lats
# longs
```

Remember to always use meaningful names for your files and array keys. It'll make your life much easier when you come back to your project later.

Both the `.npy` and `.npz` formats are specific to NumPy. If you need to share your data with someone not using NumPy, consider using more universal formats like CSV.

A good feature about working with `np.load()` is that it doesn't actually read the data into memory until you access it. This is great for working with large datasets that might not fit into your computer's memory all at once.

Basic linear algebra operations

Let's start with two fundamental operations: dot product and matrix multiplication. These might sound similar, but they have distinct uses and implementations in NumPy.

Dot Product (`np.dot()`)

The dot product is a way to multiply two arrays or vectors. It's super useful when you're working with things like GPS coordinates or when you need to calculate distances or similarities between points.

Here's how it works:

```
import numpy as np

# Let's say we have two GPS coordinates (latitude, longitude)
point_A = np.array([40.7128, -74.0060]) # New York City
point_B = np.array([34.0522, -118.2437]) # Los Angeles

# Calculate the dot product
dot_product = np.dot(point_A, point_B)

print(f"The dot product is: {dot_product}")
```

In this example, we're not really calculating anything meaningful geographically, but it shows how to use `np.dot()`. The dot product gives us a single number that represents a kind of relationship between these two points.

You might use the dot product when implementing algorithms for nearest neighbour searches or when calculating the cosine similarity between vectors.

Matrix Multiplication (@ operator)

Matrix multiplication is like the dot product's more complex cousin. It's incredibly useful when you're working with multiple sets of coordinates or when you need to transform spatial data.

```
import numpy as np

# Let's create a simple transformation matrix (rotation)
rotation_matrix = np.array([[0, -1],
                             [1, 0]]) # 90-degree rotation

# And a set of coordinates
coordinates = np.array([[1, 2],
                        [3, 4],
                        [5, 6]]) # Three points (x, y)

# Apply the rotation to our coordinates
rotated_coordinates = coordinates @ rotation_matrix

print("Original coordinates:")
print(coordinates)
print("\nRotated coordinates:")
print(rotated_coordinates)
```

In this example, we're rotating our coordinates by 90 degrees. You can check out the maths in your old high school textbooks if you don't believe me. The @ operator makes this matrix multiplication super easy and readable.

Introduction to structured arrays

Now, you might be wondering, "What exactly is a structured array, and why should I care?" In simple terms, a structured array is a fancy way of organizing data in NumPy that allows you to store different types of data in a single array. It's like having a spreadsheet where each column can have its own data type. This is particularly useful when dealing with complex data sets, where you might have a mix of numbers, text, and even dates all in one place.

Let's start by creating a structured array. Imagine we're working with data about cities, including their names, populations, and coordinates (can you tell yet that I'm a professor in Spatial Sciences?). Here's how we can create a structured array to hold this information:

```
import numpy as np

# Define the structure of our array
city_dtype = np.dtype([('name', 'U20'), ('population', 'i4'), ('latitude', 'f4'), ('longitude', 'f4')])

# Create the structured array
cities = np.array([
    ('New York', 8419000, 40.7128, -74.0060),
    ('Tokyo', 13960000, 35.6762, 139.6503),
    ('Paris', 2161000, 48.8566, 2.3522)
], dtype=city_dtype)

print(cities)
```

In this example, we're defining a custom data type (dtype) that specifies the structure of our array. Each field has a name and a type:

- █ 'U20' means a Unicode string of up to 20 characters
- █ 'i4' is a 32-bit integer
- █ 'f4' is a 32-bit float

When we create the array, we pass in our data and specify the dtype we just created. This tells NumPy how to interpret and store the data.

Basic array manipulation

Imagine you're working with a dataset of geographical coordinates. Sometimes, you might need to switch the arrangement of your data from rows to columns or vice versa. This is where transposing comes in handy. In NumPy, there are two main ways to transpose an array:

Using the T attribute:

This is the quickest and easiest way to transpose a 2D array.

Let's say we have an array of latitude and longitude coordinates:

```
import numpy as np

coords = np.array([[40.7128, -74.0060], # New York
                  [51.5074, -0.1278], # London
                  [35.6762, 139.6503]]) # Tokyo

print("Original array:")
print(coords)

print("\nTransposed array:")
print(coords.T)
```

Output:

```
Original array:
[[ 40.7128 -74.006 ]
 [ 51.5074 -0.1278]
 [ 35.6762 139.6503]]
```

```
Transposed array:
[[ 40.7128  51.5074  35.6762]
 [ -74.006  -0.1278 139.6503]]
```

As you can see, our original array had each city's coordinates as a row. After transposing, we now have latitudes in the first row and longitudes in the second.

The T attribute is a view of the original array, not a copy. This means it's memory-efficient, but any changes to the transposed view will affect the original array.

Using `np.transpose(array, axes=None)`:

This method is more flexible and can handle arrays with more than two dimensions.

```
print("Using np.transpose():")
print(np.transpose(coords))
```

The output will be the same as using the T attribute in this case.

Transposing is particularly useful when you need to apply operations across a different axis of your data, or when preparing data for certain plotting functions. If the axes are given, as a tuple, their order indicates how the axes are permuted

Now, let's say you want to reverse the order of your data. Maybe you want to list your cities from east to west instead of west to east. For this, you can use NumPy's `flip(array, axis)` function.

```
print("Original array:")
print(coords)

print("\nReversed array (along rows):")
print(np.flip(coords, axis=0))

print("\nReversed array (along columns):")
print(np.flip(coords, axis=1))
```

Output:

```
Original array:
[[ 40.7128 -74.006 ]
 [ 51.5074 -0.1278]
 [ 35.6762 139.6503]]

Reversed array (along rows):
[[ 35.6762 139.6503]
 [ 51.5074 -0.1278]
 [ 40.7128 -74.006 ]]

Reversed array (along columns):
[[ -74.006  40.7128]
 [ -0.1278 51.5074]
 [ 139.6503 35.6762]]
```

When we reverse along axis 0 (rows), we're changing the order of our cities. When we reverse along axis 1 (columns), we're swapping longitude and latitude for each city.

NumPy ecosystem

Before we dive deeper into NumPy itself, let's take a quick look at the broader ecosystem it's a part of:

- 🟩 SciPy: Think of SciPy as NumPy's big brother. It builds on top of NumPy and provides additional functionality for scientific and technical computing. We will dive into SciPy when we look at machine learning.
- 🟩 Pandas: If NumPy is great for working with numerical arrays, Pandas is fantastic for working with tabular data. We will cover Pandas after our look at NumPy.
- 🟩 Matplotlib: This is your visualization powerhouse. It is a perfectly library for visualising your data and plot beautiful maps. We will dedicate four lessons to Matplotlib alone.

These libraries work seamlessly together, forming a powerful toolkit for data analysis. But for now, we're focusing on the cornerstone of this ecosystem: NumPy.

If you would like more information on the methods and concepts that we've covered so far, please check out the excellent official Documentation: <https://numpy.org/doc/>. It is comprehensive and well-organized.



Array Operations

We'll start this section with a brief revisit of array shape manipulation, looking at reshaping, flattening, and expanding arrays. The `reshape()` function is your go-to tool for changing the shape of an array without altering its data. Here, we've taken our 1D array and reshaped it into a 3x4 grid.

```
import numpy as np

elevations = np.array([10, 20, 30, 40, 50, 60, 70, 80,
                      90, 100, 110, 120])
print(elevations.shape) # Output: (12,)

reshaped = elevations.reshape(3, 4)
print(reshaped)
# Output:
# [[ 10  20  30  40]
# [ 50  60  70  80]
# [ 90 100 110 120]]
```

Remember, the total number of elements must remain the same. You can also use `-1` to let NumPy automatically calculate one dimension:

```
auto_reshaped = elevations.reshape(3, -1)
print(auto_reshaped.shape) # Output: (3, 4)
```

■ `resize()` function

While `reshape()` returns a new array, `resize()` modifies the array in-place. It can also change the total number of elements, either by truncating or repeating data.

```
coordinates = np.array([[1, 2], [3, 4], [5, 6]])
print(coordinates)
# Output:
# [[1 2]
# [3 4]]
```

```
# [5 6]]

coordinates.resize(2, 2)
print(coordinates)
# Output:
# [[1 2]
# [3 4]]

coordinates.resize(4, 2, refcheck=False)
print(coordinates)
# Output:
# [[1 2]
# [3 4]
# [1 2]
# [3 4]]
```

We use `refcheck=False` to allow resizing even if other variables reference the array. Be cautious with this!

■ `flatten()` vs `ravel()` - flattening arrays

Both `flatten()` and `ravel()` convert multi-dimensional arrays to 1D, but with a key difference. Flatten always returns a copy of the array, while `ravel` will return a view if possible, otherwise a copy.

```
terrain = np.array([[100, 200, 300], [400, 500, 600]])

flat_copy = terrain.flatten() # Always returns a copy
flat_view = terrain.ravel()   # Returns a view if
                              # possible, else a copy

flat_copy[0] = 999
flat_view[0] = 999

print(terrain) # Original unchanged by flatten()
# Output:
# [[100 200 300]
# [400 500 600]]

print(terrain) # Original changed by ravel()
# Output:
# [[999 200 300]
# [400 500 600]]
```

Use `flatten()` when you want to ensure you're working with a new, independent copy. Use `ravel()` for better performance when you're okay with modifying the original data.

■ `expand_dims()` - adding dimensions

Sometimes, you need to add a new axis to your array, especially when working with libraries that expect a certain number of dimensions.

```
lat_long = np.array([34.0522, -118.2437]) # Los Angeles
coordinates
print(lat_long.shape) # Output: (2,)

expanded = np.expand_dims(lat_long, axis=0)
print(expanded.shape) # Output: (1, 2)
print(expanded)
# Output: [[ 34.0522 -118.2437]]
```

This is particularly useful when you need to add a batch dimension for machine learning models or when working with time series data.

■ `squeeze()` - removing single-dimensional entries

`np.squeeze()` is the opposite of `expand_dims()`, removing all single-dimensional entries from the shape of an array. It removes single-dimensional entries, or axes, from the shape of an array. For example, if you have a 3D array with a shape of (1, 5, 1), `np.squeeze()` will remove the dimensions of size 1, resulting in a shape of (5,). This is useful when you want to simplify the dimensions of an array, especially after operations that add unnecessary dimensions, such as when indexing or using certain functions that return arrays with extra dimensions. It helps in making the array easier to work with by reducing its dimensionality without altering the data.

```
rainfall = np.array([[5], [10]], [[15], [20]])
print(rainfall.shape) # Output: (2, 2, 1)

squeezed = np.squeeze(rainfall)
print(squeezed.shape) # Output: (2, 2)
print(squeezed)
# Output:
# [[ 5 10]
#  [15 20]]
```

This is handy when you've ended up with unnecessary dimensions after operations like slicing or expanding.

Indexing techniques (revisited)

We looked at indexing techniques in the previous section, especially boolean indexing and fancy indexing. These techniques are fundamental to using NumPy in data science, so it is worth having another closer look at the techniques.

Boolean indexing is like using a filter on your data. Imagine you have a satellite image of a landscape, and you want to focus only on areas above a certain elevation. That's where boolean indexing shines!

```
import numpy as np

# Let's create an array representing elevation data
elevation = np.array([[100, 200, 150],
                     [300, 250, 180],
                     [120, 280, 310]])

# We want to find all areas above 200 meters
high_ground = elevation > 200

print("Boolean mask:")
print(high_ground)

# Now, let's use this mask to get only the high
# elevation points
high_elevation = elevation[high_ground]

print("\nHigh elevation points:")
```

```
print(high_elevation)
```

In this example, `high_ground` is our boolean mask. It's an array of the same shape as `elevation`, but filled with True and False values. When we use this mask to index `elevation`, we get only the values where the mask is True. Boolean indexing is great for data cleaning. You can easily remove or replace outliers or invalid data points.

Fancy indexing allows us to cherry-pick elements from our array using another array of integers. This is very useful when you need to select specific data points based on their indices.

```
# Array of (latitude, longitude) pairs
coordinates = np.array([[-33.865, 151.209], # Sydney
                       [40.712, -74.006], # New York
                       [35.689, 139.691], # Tokyo
                       [51.507, -0.127]]) # London

# Let's say we want to select Sydney and Tokyo
city_indices = np.array([0, 2])

selected_cities = coordinates[city_indices]

print("Selected cities:")
print(selected_cities)
```

Here, `city_indices` is our integer array used for indexing. It selects the rows at indices 0 and 2 from our `coordinates` array. This is incredibly useful when you have a large dataset of data points and need to quickly extract specific items based on their index in the dataset.

The real power, however, comes when we combine these methods:

```
# Array of weather data: [temperature, humidity,
#                          wind_speed]
weather_data = np.array([[25, 60, 10],
                         [30, 55, 15],
                         [20, 70, 5],
                         [35, 50, 20],
                         [22, 65, 8]])

# We want to select humidity and wind speed for
# locations where temperature > 25
# and wind speed < 12

# First, create our boolean mask
temp_wind_mask = (weather_data[:, 0] > 25) &
(weather_data[:, 2] < 12)

# Now, use this mask along with integer indexing
selected_data = weather_data[temp_wind_mask[:, 1:]]

print("Selected weather data (humidity, wind speed):")
print(selected_data)
```

Here, we're combining boolean indexing (`temp_wind_mask`) with integer indexing (`[:, 1:]`) to select specific rows and columns based on our conditions.

Some key things to remember:

- Boolean indexing is your go-to for filtering data based on conditions.
- Fancy indexing with integer arrays is perfect for selecting specific elements or subsets of your data.
- Combining methods gives you ultimate flexibility in data selection.

Try creating your own datasets and experiment with these indexing techniques. You'll be surprised at how much you can simplify complex data operations!

Set operations

NumPy offers a range of set operations that can make your life much easier when you need to compare, combine, or find differences between datasets.

■ `unique()` - Finding Unique Elements

The `unique()` function is like a magic wand that removes all duplicate elements from an array, leaving you with only the unique values. It's super useful when you want to know what distinct values you have in your dataset.

Let's say you have an array of prices for items, and you want to know how many unique prices you're dealing with:

```
import numpy as np

prices = np.array([10.5, 12.3, 10.5, 15.7, 12.3, 18.2])
unique_prices = np.unique(prices)
print(unique_prices)
# Output: [ 10.5  12.3  15.7  18.2]
```

See how it removed the duplicates? This can be really handy when you're trying to get a quick overview of your data or when you need to remove redundant information.

■ `intersect1d()` - Finding Common Elements

Now, imagine you have two sets of coordinates and you want to find out which points they have in common. You can find out using the `intersect1d()` method.

```
coordinates_set1 = np.array([(1, 2), (3, 4), (5, 6)])
coordinates_set2 = np.array([(3, 4), (5, 6), (7, 8)])

common_coordinates = np.intersect1d(coordinates_set1,
                                     coordinates_set2)
print(common_coordinates)
# Output: [(3, 4) (5, 6)]
```

This function finds the elements that exist in both arrays.

■ `union1d()` - Combining Unique Elements

`union1d()` combines all unique elements from two arrays. It makes sure that elements are not recorded twice.

```
cities_group1 = np.array(['New York', 'London', 'Paris'])
cities_group2 = np.array(['Tokyo', 'London', 'Sydney'])

all_unique_cities = np.union1d(cities_group1,
                               cities_group2)
print(all_unique_cities)
# Output: ['London' 'New York' 'Paris' 'Sydney' 'Tokyo']
```

This is great when you want to merge datasets while avoiding duplicates.

■ `setdiff1d()` - Finding Elements in One Array but not another

`setdiff1d()` is like playing spot the difference between two datasets. It finds elements that are in the first array but not in the second.

```
all_landmarks = np.array(['Eiffel Tower', 'Big Ben', 'Statue of Liberty', 'Sydney Opera House'])
visited_landmarks = np.array(['Big Ben', 'Sydney Opera House'])

yet_to_visit = np.setdiff1d(all_landmarks,
                             visited_landmarks)
print(yet_to_visit)
# Output: ['Eiffel Tower' 'Statue of Liberty']
```

This can be very useful when you're trying to identify what's missing or what's extra in one dataset compared to another.

■ `setxor1d()` - Finding Elements in Either Array but Not Both

Last but not least, `setxor1d()` finds elements that are in either of the arrays, but not in both. It's like finding the unique elements in each dataset.

```
team1_routes = np.array(['A', 'B', 'C', 'D'])
team2_routes = np.array(['B', 'D', 'E', 'F'])

unique_routes = np.setxor1d(team1_routes, team2_routes)
print(unique_routes)
# Output: ['A' 'C' 'E' 'F']
```

This is great for identifying differences between two datasets or finding elements that are unique to each set.

Remember, these functions work best with 1D arrays. If you have multi-dimensional data, you might need to flatten it first or use other NumPy functions.

Sorting arrays

NumPy, offers a variety of sorting functions that can make your life much easier when working with arrays. Whatever type of data you are working with, knowing how to efficiently sort your data is essential.

■ The `sort(array, axis=-1, kind)` function

Let's start with the basic `sort()` function. This is your go-to tool for simple sorting tasks.

The `sort()` function arranges the elements of an array in ascending order by default. You can use the `axis` parameter to tell it over which axis of your array you would like to sort. By default, it sorts along the last axis.

```
import numpy as np

elevations = np.array([1200, 800, 1500, 950, 1100])
elevations.sort()
print("Sorted elevations:", elevations)
```

Output:

```
Sorted elevations: [ 800  950 1100 1200 1500]
```

You can also indicate what kind of sort algorithm you would like to use. Possible options include `'quicksort'`, `'mergesort'`, and `'heapsort'`. To see a more complete list try to use `help(np.sort)`.

■ `argsort()` - Getting Sorted Indices

Now, what if you want to know the original positions of the sorted elements? That's where `argsort()` comes in handy! `argsort()` returns the indices that would sort an array without sorting the array.

```
coordinates = np.array([(2, 3), (1, 2), (3, 1), (2, 2)])
sorted_indices = coordinates.argsort(axis=0)
print("Sorted indices:", sorted_indices)
print("Sorted coordinates:",
      coordinates[sorted_indices[:, 0]])
```

Output:

```
Sorted indices: [[1 2]
                [0 3]
                [3 1]
                [2 0]]
Sorted coordinates: [[1 2]
                    [2 2]
                    [2 3]
                    [3 1]]
```

■ `lexsort(keys)` - Sorting by Multiple Keys

`np.lexsort()` performs an indirect sort using a sequence of keys provides as a tuple parameter. You would use `np.lexsort(keys)` when you need to sort

an array by multiple keys (columns) in a specific order. This function returns an array of indices that sorts the data according to the sequence of keys provided. It's particularly useful when you want to perform a stable, multi-level sort, where the first key is the primary sort criterion, the second key is the secondary criterion, and so on.

```
countries = np.array(['USA', 'France', 'USA',
                     'Germany'])
populations = np.array([1000000, 500000, 750000,
                       600000])

sorted_indices = np.lexsort((populations, countries))
print("Sorted data:")
for i in sorted_indices:
    print(f"{countries[i]}: {populations[i]}")
```

This will output:

```
Sorted data:
France: 500000
Germany: 600000
USA: 750000
USA: 1000000
```

The keys in `lexsort()` are applied from last to first. So, the last key in the tuple is the primary sort key.

■ `searchsorted()` - Finding Insertion Points

Last but not least, we have `searchsorted()`. This function is great when you're working with already sorted data. `searchsorted()` finds the indices where elements should be inserted to maintain order in a sorted array.

```
sorted_elevations = np.array([800, 950, 1100, 1200,
                             1500])
new_elevation = 1300
insert_position = np.searchsorted(sorted_elevations,
                                  new_elevation)
print(f"Insert {new_elevation} at position:
      {insert_position}")

# Output: Insert 1300 at position: 4
```

When updating a database, `searchsorted()` can help you quickly find where to insert new data points while maintaining the sorted order.

Array comparison operations

In NumPy, we can compare entire arrays element by element using comparison operators. This is very handy when you want to check conditions across your entire dataset at once.

Let's say we have two arrays representing elevations in meters for two different mountain ranges:

```
import numpy as np

range1 = np.array([1000, 1500, 2000, 2500, 3000])
range2 = np.array([1200, 1400, 2100, 2300, 2800])
```

We can compare these arrays using the following operators: `==` (equal to), `!=` (not equal to), `<` (less than), `>` (greater than), `<=` (less than or equal to), `>=` (greater than or equal to). For example, let's check which elevations in `range1` are higher than those in `range2`:

```
higher_than_range2 = range1 > range2
print(higher_than_range2)
# Output: [False True False True True]
```

This gives us a boolean array where True indicates that the elevation in `range1` is higher than the corresponding elevation in `range2`. You could then apply this mask to another array.

Sometimes, we want to know if all elements in an array meet a certain condition, or if any of them do. That's where the `all()` and `any()` functions come in handy.

Let's check if all elevations in `range1` are above 1000 meters:

```
all_above_1000 = np.all(range1 > 1000)
print(all_above_1000) # Output: True
```

Now, let's see if any elevations in `range2` are above 3000 meters:

```
any_above_3000 = np.any(range2 > 3000)
print(any_above_3000) # Output: False
```

These functions are great for quick checks on your data!

The `where()` function is like a vectorized if-else statement. It's incredibly useful for selecting elements based on conditions and very fast.

Imagine we want to create a new array where we take the higher elevation from each pair in `range1` and `range2`:

```
higher_elevations = np.where(range1 > range2, range1,
                             range2)
print(higher_elevations)
# Output: [1200 1500 2100 2500 3000]
```

This says: "Where `range1` is greater than `range2`, use the value from `range1`; otherwise, use the value from `range2`."

Let's apply this to a real-world scenario. Suppose we have two arrays representing annual rainfall (in mm)

for two regions over five years and we wanted to know how the two compare:

```
region1_rainfall = np.array([800, 950, 1100, 1000,
                             1200])
region2_rainfall = np.array([750, 1000, 1050, 1150,
                             1100])

# Find years where region1 had more rainfall
wetter_years_region1 = region1_rainfall >
region2_rainfall
print("Years where Region 1 had more rainfall:",
      wetter_years_region1)

# Check if Region 2 ever had rainfall above 1100mm
region2_heavy_rainfall = np.any(region2_rainfall > 1100)
print("Did Region 2 ever have rainfall above 1100mm?",
      region2_heavy_rainfall)

# Create an array with the higher rainfall value for
each year
higher_rainfall = np.where(region1_rainfall >
region2_rainfall, region1_rainfall, region2_rainfall)
print("Higher rainfall values:", higher_rainfall)
```

Logical operations on arrays

NumPy has three main logical functions that you can use to combine or negate conditions across entire arrays in one go: `logical_and()`, `logical_or()`, and `logical_not()`.

■ `np.logical_and()`:

This function performs an element-wise AND operation. It returns True only if both input conditions are True.

■ `np.logical_or()`:

The counterpart to the `logical_and()` function, the `logical_or()` function returns True if either of the input conditions is True or False otherwise.

■ `np.logical_not()`:

This function inverts the boolean values in an array. As simple as that.

```
# Find areas that are NOT high elevation
not_high = np.logical_not(elevation > 150)
print(not_high)
# Output: [ True False  True False  True]
```

Imagine you're analysing land cover data and want to identify areas suitable for a certain type of agriculture. You might use `logical_and()` to find areas with the right combination of elevation, rainfall, and soil type.

Array iteration

When working with NumPy arrays, especially large ones, you'll often need to process each element individually. This is where array iteration comes in

handy. NumPy provides powerful tools to make this process smooth and efficient. We'll focus on two key concepts: the `nditer` object and `ndenumerate`.

■ The `nditer` object:

The `nditer` (short for N-dimensional iterator) is a flexible tool for iterating over NumPy arrays. It's like a version of a regular Python loop, designed specifically for NumPy arrays.

Let's start with a simple example using our trusted example of elevation data:

```
import numpy as np

# Creating a 2D array representing elevation data (in meters)
elevation = np.array([[100, 200, 150],
                     [120, 180, 160],
                     [90, 220, 140]])

print("Elevation data:")
print(elevation)

print("\nIterating through elevation data:")
for element in np.nditer(elevation):
    print(element, end=' ')
```

This will output:

```
Elevation data:
[[100 200 150]
 [120 180 160]
 [ 90 220 140]]

Iterating through elevation data:
100 200 150 120 180 160 90 220 140
```

As you can see, `nditer` automatically flattens the 2D array and allows us to iterate through all elements easily.

But wait, there's more! The `nditer` object is incredibly flexible. You can control the iteration order, modify values during iteration, and even iterate over multiple arrays simultaneously.

Here's an example where we convert elevation data from meters to feet:

```
print("\nConverting elevation from meters to feet:")
with np.nditer(elevation, op_flags=['readwrite']) as it:
    for x in it:
        x[...] = x * 3.28084 # Convert meters to feet

print(elevation)
```

Output:

```
Converting elevation from meters to feet:
[[328.084  656.168  492.126 ]
 [393.7008 590.5512 524.9344 ]
 [295.2756 721.7848 459.3176 ]]
```

The `'readwrite'` flag allows us to modify the array in-place, which can be much more memory-efficient for large datasets.

■ `ndenumerate` for index-value pairs:

Sometimes, you need to know not just the value of each element, but also its position in the array. This is where `ndenumerate` comes in handy, just like the `enumerate` function with standard Python loops. It returns both the index and value for each element.

```
# Creating a 2D array representing (longitude, latitude) pairs
coordinates = np.array([[ -74.006, 40.7128], # NYC
                       [-87.6298, 41.8781], # Chicago
                       [-118.2437, 34.0522]]) # LA

print("Coordinates data:")
print(coordinates)

print("\nIterating through coordinates with indices:")
for index, value in np.ndenumerate(coordinates):
    print(f"Index: {index}, Value: {value}")
```

Here's the output:

```
Coordinates data:
[[ -74.006  40.7128]
 [ -87.6298  41.8781]
 [-118.2437  34.0522]]

Iterating through coordinates with indices:
Index: (0, 0), Value: -74.006
Index: (0, 1), Value: 40.7128
Index: (1, 0), Value: -87.6298
Index: (1, 1), Value: 41.8781
Index: (2, 0), Value: -118.2437
Index: (2, 1), Value: 34.0522
```

This is particularly useful when you need to perform operations based on both the value and its position in the array.

Let's combine what we've learned to solve a real-world problem. Imagine we have temperature data for several cities, and we want to convert temperatures above 30°C to Fahrenheit:

```
# Temperature data for cities (in Celsius)
temperatures = np.array([[28, 32, 25],
                         [33, 29, 35],
                         [30, 31, 26]])

city_names = ['New York', 'Los Angeles', 'Chicago']

print("Original temperature data (°C):")
print(temperatures)

for index, temp in np.ndenumerate(temperatures):
    if temp > 30:
        # Convert to Fahrenheit if above 30°C
        temperatures[index] = temp * 9/5 + 32
        print(f"Converted {city_names[index[0]]}
        temperature at index {index} to
        {temperatures[index]:.1f}°F")

print("\nUpdated temperature data:")
print(temperatures)
```

Use `nditer` when you need to iterate over all elements efficiently, especially for multi-dimensional arrays. But use `ndenumerate` when you need both index and value information. Remember that `nditer` flattens the array by default, but you can control the iteration order if needed.

Array manipulation using ufuncs

We already had a look at ufuncs in the previous section, here is a more detailed look at them, including how to create your own. Remember, ufuncs operate element-by-element on arrays, allowing you to perform calculations across entire datasets in one go.

Let's start with some built-in ufuncs that you'll use frequently:

Basic arithmetic operations:

NumPy provides ufuncs for addition, subtraction, multiplication, and division. These are super useful for operations like scaling or normalizing data.

```
import numpy as np

# Let's say we have an array of elevation data in meters
elevations = np.array([100, 150, 200, 250, 300])

# We want to convert it to feet (1 meter = 3.28084 feet)
elevations_feet = np.multiply(elevations, 3.28084)

print(elevations_feet)
# Output: [328.084 492.126 656.168 820.21 984.252]
```

Trigonometric functions:

These are crucial for many geospatial calculations, especially when working with coordinates. We have ufuncs like `np.sin()`, `np.cos()`, `np.tan()`, ...

```
# Converting degrees to radians
degrees = np.array([0, 30, 45, 60, 90])
radians = np.radians(degrees)

# Calculating sine values
sin_values = np.sin(radians)

print(sin_values)
# Output: [0.         0.5        0.70710678 0.8660254
1.         ]
```

Exponential and logarithmic functions:

These come in handy for various scaling operations and statistical analyses.

```
# Calculating exponential growth
time = np.array([0, 1, 2, 3, 4])
growth_rate = 0.1
population = np.exp(growth_rate * time)

print(population)
```

```
# Output: [1.         1.10517092 1.22140276 1.34985881
1.4918247 ]
```

While NumPy's built-in ufuncs are great, sometimes you need something tailored to your specific needs. That's why you can create your own ufuncs too.

Using the `np.frompyfunc()` method allows you to create a ufunc from any Python function. It's flexible but might not be as fast as other methods.

```
def celsius_to_fahrenheit(celsius):
    return (celsius * 9/5) + 32

# Create a ufunc
celsius_to_fahrenheit_ufunc =
np.frompyfunc(celsius_to_fahrenheit, 1, 1)

# Use it on an array of temperatures
celsius_temps = np.array([-10, 0, 10, 20, 30])
fahrenheit_temps =
celsius_to_fahrenheit_ufunc(celsius_temps)

print(fahrenheit_temps)
# Output: [14.0 32.0 50.0 68.0 86.0]
```

Here, we've created a custom ufunc to convert temperatures from Celsius to Fahrenheit, which we can now use directly on our arrays.

This is another way to create a ufunc-like object, often used for its simplicity is using the `@np.vectorize()` decorator. It transforms a function that operates on scalars into a vectorized function that can apply the scalar function element-wise to arrays. This allows you to apply a function to each element of an array efficiently, without the need for explicit loops.

```
@np.vectorize
def classify_elevation(elevation):
    if elevation < 100:
        return "Low"
    elif 100 <= elevation < 500:
        return "Medium"
    else:
        return "High"

elevations = np.array([50, 200, 600, 300, 800])
classifications = classify_elevation(elevations)

print(classifications)
# Output: ['Low' 'Medium' 'High' 'Medium' 'High']
```

Record arrays

If you've been working with structured arrays, you're going to love record arrays. Record arrays take the concept of structured arrays and make them even more user-friendly and intuitive. Let's explore what they are, how to use them, and why they're particularly useful for data science.

Record arrays are essentially structured arrays with some extra bells and whistles. They allow you to

access fields using attribute notation, which makes your code more readable and intuitive. Think of them as a hybrid between NumPy arrays and Python objects.

Let's start by creating a simple record array using `np.recarray`.

```
import numpy as np

# Define the data type
dt = np.dtype([('name', 'U20'), ('lat', 'f4'), ('lon', 'f4'), ('population', 'i4')])

# Create the record array
cities = np.array([
    ('New York', 40.7128, -74.0060, 8419000),
    ('Tokyo', 35.6762, 139.6503, 13929286),
    ('Paris', 48.8566, 2.3522, 2161000)
], dtype=dt)

# Convert to record array
cities = cities.view(np.recarray)

print(cities)
```

In this example, we've created a record array with information about cities, including their names, latitudes, longitudes, and populations. With record arrays, you can access fields using dot notation:

```
print(cities.name) # Prints all city names
print(cities.lat) # Prints all latitudes
print(cities[0].population) # Prints population of the first city
```

Isn't that neat? It's like working with objects, making your code more intuitive and easier to read.

You can manipulate record arrays just like regular NumPy arrays. Let's add a new city:

```
new_city = np.array([('Sydney', -33.8688, 151.2093, 5312000)], dtype=dt).view(np.recarray)
cities = np.concatenate((cities, new_city))

print(cities)
```

Record arrays in NumPy provide a user-friendly way to work with structured data. They combine the power of NumPy arrays with the intuitive access of Python objects. Remember, they're particularly handy when you're dealing with data that has named fields and you want a more readable and maintainable code.

Array padding

Imagine you have a satellite image of a city, represented as a NumPy array. Sometimes, you might need to add extra space around this image – maybe to prepare it for a specific algorithm or to make it fit a certain size. That's where padding comes in handy!

NumPy provides us with a fantastic function for this purpose: `numpy.pad()`. This function allows us to add values around the edges of our arrays in various ways. It's like giving your array a little buffer zone!

```
numpy.pad(array, pad_width, mode='constant', **kwargs)
```

Here, `'array'` is our input array, `'pad_width'` tells NumPy how much padding to add, and `'mode'` determines how we fill this padding. There are several modes available, and we'll explore some of the most useful ones.

Let's start with a simple example. Say we have a 3x3 array representing elevation data for a small area:

```
import numpy as np

elevation = np.array([[100, 110, 120],
                      [105, 115, 125],
                      [110, 120, 130]])

print("Original elevation data:")
print(elevation)
```

Now, let's add some padding around this data:

```
padded_elevation = np.pad(elevation, pad_width=1,
                           mode='constant', constant_values=0)

print("\nPadded elevation data:")
print(padded_elevation)
```

In this example, we've added a border of zeros around our elevation data. This 'constant' mode is one of the simplest padding methods by adding a constant value around the edges of the array.

But what if we want our padding to be more meaningful?

■ 'edge' mode:

This mode extends the edge values of the array. It's useful when you want your padding to be a continuation of your data.

```
edge_padded = np.pad(elevation, pad_width=1,
                      mode='edge')
print("\nEdge-padded elevation data:")
print(edge_padded)
```

■ 'mean' mode:

This mode uses the mean of all values in the array for padding. It's great when you want your padding to represent an average of your data.

```
mean_padded = np.pad(elevation, pad_width=1,
                      mode='mean')
print("\nMean-padded elevation data:")
print(mean_padded)
```

■ 'reflect' mode:

This mode reflects the values at the edge of the array. It's like creating a mirror image of your data at the edges.

```
reflect_padded = np.pad(elevation, pad_width=1,
mode='reflect')
print("\nReflect-padded elevation data:")
print(reflect_padded)
```

These are just a few of the many padding modes available. Each mode can be useful in different scenarios, depending on your data and what you're trying to achieve. Make sure to look into the documentation for all the different modes.

Pro tip: When working with multi-dimensional arrays, you can specify different padding for each dimension. For example:

```
# Different padding for rows and columns
custom_padded = np.pad(elevation, pad_width=((1,2),
(3,1)), mode='constant', constant_values=0)
print("\nCustom-padded elevation data:")
print(custom_padded)
```

This adds 1 row at the top, 2 at the bottom, 3 columns on the left, and 1 on the right.

Remember, the key to mastering array padding is understanding your data and choosing the appropriate padding mode. Don't be afraid to experiment with different modes to see which works best for your specific use case.

Rolling window operations

NumPy offers rolling window operations and some cool tricks with `strides`! This is where we start to see how NumPy can really turbocharge our data analysis, especially when working with time series or spatial data.

Imagine you're looking at a long street through a small window in a moving car. As you move, you see different parts of the street. That's essentially what a rolling window does with data - it "slides" over your dataset, allowing you to perform operations on subsets of your data. In geospatial science, you might use them to calculate moving averages of temperature data, smooth out noise in satellite imagery, or detect patterns in land use changes over time.

NumPy has a powerful function called `as_strided()`. It's great for creating views of your data, including rolling windows.

Here's a simple example:

```
import numpy as np
from numpy.lib.stride_tricks import as_strided

# Let's say we have elevation data for a hiking trail
elevation_data = np.array([100, 120, 115, 130, 140, 135,
150, 160])

# We want to calculate a moving average with a window
size of 3
window_size = 3

# Here's where the magic happens
windowed_data = as_strided(elevation_data,
                           shape=(len(elevation_data) -
window_size + 1, window_size),

                           strides=(elevation_data.strides[0],
elevation_data.strides[0]))

print(windowed_data)
# Output:
# [[100 120 115]
#  [120 115 130]
#  [115 130 140]
#  [130 140 135]
#  [140 135 150]
#  [135 150 160]]

# Now we can easily calculate the moving average
moving_average = np.mean(windowed_data, axis=1)
print(moving_average)
# Output: [111.66666667 121.66666667 128.33333333 135.
141.66666667 148.33333333]
```

In this example, we're creating a rolling window view of our elevation data. Each row in `windowed_data` represents a window of three consecutive elevation points. We then calculate the moving average for each window.

So, what's going on under the hood with `as_strided()`? It's all about strides! In NumPy, strides are the number of bytes to step in each dimension when traversing an array. By manipulating these strides, we can create different views of our data without actually copying it.

Here's a simple visualization:

```
Original data: [100, 120, 115, 130, 140, 135, 150, 160]
               ^   ^   ^
               |   |   |
Window 1:      [100, 120, 115]
               ^   ^   ^
               |   |   |
Window 2:           [120, 115, 130]
               ^   ^   ^
               |   |   |
Window 3:               [115, 130, 140]
... and so on
```

Each window is just a view into the original data, created by clever use of strides.

Binning operations

Alright, let's dive into another exciting part of NumPy – binning operations! This is a crucial topic for anyone working with data analysis.

Binning, also known as discretization, is the process of grouping continuous data into discrete intervals or "bins". It's like sorting items into different boxes based on their characteristics.

NumPy provides powerful tools for binning operations. We'll focus on two main functions: `digitize()` and `histogram()/histogram2d()`.

■ The `digitize()` function

Let's start with the `digitize()` function. This nifty tool assigns each value in an input array to a bin, based on the bin edges you specify.

Here's how it works:

```
import numpy as np

# Let's say we have elevation data (in meters) for a region
elevations = np.array([10, 45, 75, 120, 150, 200, 250, 300])

# We want to categorize these into bins: 0-50m, 51-100m, 101-200m, 201m+
bin_edges = np.array([0, 50, 100, 200, np.inf])

# Use digitize to assign each elevation to a bin
binned_elevations = np.digitize(elevations, bin_edges)

print(binned_elevations)
```

This will output: [1 1 2 3 3 3 4 4]

What does this mean? The first two elevations (10m and 45m) fall into the first bin (0-50m), the 75m elevation falls into the second bin (51-100m), and so on.

The bins in `digitize()` are left-edge inclusive and right-edge exclusive, except for the last bin which includes both edges.

■ The `histogram()` and `histogram2d()` functions

The `histogram()` methods are great for understanding the distribution of your data, just like in statistics class.

```
# Using our elevation data from before
hist, bin_edges = np.histogram(elevations, bins=4)

print("Histogram:", hist)
print("Bin edges:", bin_edges)
```

This might output:

```
Histogram: [2 1 3 2]
Bin edges: [ 10.   82.5  155.  227.5  300. ]
```

This tells us we have 2 values in the first bin, 1 in the second, 3 in the third, and 2 in the fourth. The first bin is from 10 to 82.5 (excluded), the second from 82.5 to 155 (excluded), the third from 155 to 227.5 (excluded) and the final one from 227.5 to 300 (inclusive).

The `histogram2d()` function is similar, but for 2D data:

```
# Let's say we have latitude and longitude data
latitudes = np.array([30, 31, 32, 33, 34, 35])
longitudes = np.array([-100, -99, -98, -97, -96, -95])

hist, x_edges, y_edges = np.histogram2d(latitudes,
longitudes, bins=(3,3))

print("2D Histogram:\n", hist)
print("Latitude bin edges:", x_edges)
print("Longitude bin edges:", y_edges)
```

This creates a 3x3 grid over our lat-long data and counts how many points fall into each cell. This is fantastic for creating heat maps which you could use to visualize the density of certain features.

When using `histogram2d()`, remember that the first array you pass in (latitudes in our example) corresponds to the rows of the resulting 2D histogram, while the second array (longitudes) corresponds to the columns.

Try playing around with these functions using different datasets. You might be surprised at the patterns you can uncover.

Array masks and masked arrays

Imagine you have a dataset of temperatures across a country, but some weather stations didn't report their data. Instead of filling these gaps with zeros or some arbitrary value, we can use masked arrays to tell Numpy, "Hey, this value is missing or invalid, please don't include it in calculations."

■ Creating and Using Masked Arrays:

Let's start with a simple example. Say we have temperature data for a week, but we're missing data for Wednesday.

```
import numpy as np
import numpy.ma as ma

# Our temperature data (in Celsius)
temperatures = np.array([25, 28, 30, -999, 27, 29, 26])

# Create a mask (True where data is invalid)
mask = temperatures == -999
```

```
# Create a masked array
masked_temps = ma.masked_array(temperatures, mask)

print(masked_temps)
# Output: [25 28 30 -- 27 29 26]

# Now when we calculate the mean, it ignores the masked
value
print(masked_temps.mean())
# Output: 27.5
```

In this example, we used -999 as a placeholder for missing data. The mask is a boolean array where True indicates invalid data. When we create the masked array, Numpy knows to ignore that value in calculations.

Numpy's `ma.masked_where()` function is another way to create masked arrays based on a condition:

```
# Let's say we want to mask all temperatures above 28°C
high_temp_mask = ma.masked_where(temperatures > 28,
temperatures)
print(high_temp_mask)
# Output: [25 28 -- -- 27 -- 26]
```

This is really useful when you want to focus on specific ranges of data.

You can easily switch between masked and regular arrays using the `.filled()` method. This replaces masked values with a specified value:

```
filled_elevation =
masked_elevation.filled(fill_value=-9999)
```

Array input/output operations

In the previous section we looked at loading and saving NumPy arrays directly. Here we will explore how to efficiently read, write, and manipulate other data formats.

■ Text Files: `loadtxt()` and `savetxt()`

These functions are great for handling human-readable data files, like CSV (Comma-Separated Values) or simple text files. `loadtxt()` reads data from a text file and creates a NumPy array from it.

Let's say you have a file called 'elevation_data.txt' with elevation values for a terrain:

```
import numpy as np

# Reading the file
elevation = np.loadtxt('elevation_data.txt')
print(elevation)
```

This might output something like:

```
[[100 120 115]
 [110 125 118]]
```

```
[105 122 120]]
```

You can customize how `loadtxt()` reads your file. For example, if your data is comma-separated:

```
elevation = np.loadtxt('elevation_data.txt',
delimiter=',')
```

`savetxt()` saves a NumPy array to a text file. Don't confuse this function with the `save()` function which serializes a NumPy array in binary form.

```
# Saving processed data
processed_elevation = elevation * 2 # Just doubling
values for this example
np.savetxt('processed_elevation.txt',
processed_elevation)
```

■ Binary Files: `load()` and `save()`

Binary files are more efficient for large datasets, as they're faster to read and write and take up less space. As a recap to the last section, we can load and save individual arrays or collections of arrays with `load()` and `save()`.

```
# Saving elevation data in binary format
np.save('elevation_data.npy', elevation)

# Loading the saved binary file
loaded_elevation = np.load('elevation_data.npy')
print(loaded_elevation)
```

■ Memory-Mapped Files: `mmap()`

Memory-mapping is a powerful technique for working with very large files that don't fit into your computer's RAM. It allows you to work with the data as if it were in memory, but it's actually stored on disk.

```
# Creating a memory-mapped file
mmap_file = np.memmap('large_elevation_data.dat',
dtype='float32', mode='w+', shape=(1000, 1000))

# Now you can work with mmap_file as if it were a normal
NumPy array
mmap_file[0, 0] = 100
mmap_file[500, 500] = 150

# Changes are automatically synced to the file
del mmap_file # This ensures all changes are written to
disk
```

If you're working with a massive digital elevation model that doesn't fit in memory, `memmap()` allows you to process it in chunks without loading the entire file.

Tips: If you're working with mixed data types structured arrays in NumPy can be very useful. Remember that binary formats like `.npy` are not human-readable. Always keep a small sample of your data in a text format for quick inspection. Finally,

when saving data, consider including metadata like coordinate reference systems or data units to make your files more self-documenting.

Date and time operations

NumPy provides powerful tools for working with dates and times, which are essential when dealing with temporal data. In geospatial analysis, we often need to handle timestamps for satellite imagery, track changes over time, or analyse seasonal patterns. NumPy's date and time functionalities make these tasks much easier.

■ Working with datetime64 dtype

The star of the show in NumPy's date and time operations is the `datetime64` dtype. This special data type allows us to represent dates and times with high precision. Let's see how we can create a `datetime64` array:

```
import numpy as np

# Create a datetime64 array
dates = np.array(['2023-05-15', '2023-05-16', '2023-05-17'], dtype='datetime64')
print(dates)

# Output: ['2023-05-15' '2023-05-16' '2023-05-17']
```

You can specify the precision of `datetime64` by adding a unit, like `'datetime64[D]'` for days or `'datetime64[s]'` for seconds. Have a look at the [help](#) documentation for more details on precisions.

■ Date/Time Universal Functions (ufuncs)

NumPy also provides various ufuncs that work with `datetime64` arrays. These allow us to perform operations across entire arrays efficiently.

● Adding time deltas:

```
import numpy as np

dates = np.array(['2023-05-15', '2023-05-16', '2023-05-17'], dtype='datetime64')
days_to_add = np.array([1, 2, 3])

new_dates = dates + days_to_add

print(new_dates)

#Output: ['2023-05-16' '2023-05-18' '2023-05-20']
```

This is useful when you need to project dates into the future.

● Subtracting dates:

```
import numpy as np
```

```
start_date = np.datetime64('2023-01-01')
end_dates = np.array(['2023-05-15', '2023-07-01', '2023-12-31'], dtype='datetime64')

days_passed = end_dates - start_date

print(days_passed)

#Output: [134 181 364]
```

This operation is handy when calculating the number of days between events.

● Utility Functions

NumPy also provides utility functions for working with dates and times.

a) **`np.busday_count`**: Counts the number of business days between two dates.

```
import numpy as np

start = np.datetime64('2023-01-01')
end = np.datetime64('2023-12-31')

business_days = np.busday_count(start, end)
print(f"Number of business days in 2023: {business_days}")

# Output: Number of business days in 2023: 260
```

b) **`np.is_busday`**: Determines whether dates are business days.

```
import numpy as np

dates = np.array(['2023-05-15', '2023-05-20', '2023-05-21'], dtype='datetime64')
is_workday = np.is_busday(dates)

print(is_workday)

# Output: [ True False False]
```

This can be helpful when filtering data for weekday-only analysis. How cool is this?

Useful tips:

- 📌 Remember that NumPy's `datetime64` uses the Gregorian calendar, which might not be suitable for all historical analyses.
- 📌 When working with different time zones, it's often easier to convert all times to UTC before performing operations.
- 📌 For more complex date/time operations, you might want to use the `datetime` module from Python's standard library in conjunction with NumPy.

NumPy's date and time capabilities are powerful tools for handling temporal data. They allow you to efficiently work with large datasets, perform calculations, and analyse time-based patterns.

Error handling in NumPy operations

When we're crunching numbers things can sometimes go wrong. In NumPy, errors can occur for various reasons - division by zero, overflow (when a number gets too big for NumPy to handle), or underflow (when a number is too close to zero). By default, NumPy will raise exceptions when these errors occur, but sometimes we might want to handle these situations differently.

■ The `seterr()` Function:

The `seterr()` function is like a control panel for how NumPy should behave when it encounters different types of errors.

Let's look at a simple example:

```
import numpy as np

# Default error settings
print(np.geterr())

# Change error settings
np.seterr(divide='ignore', invalid='warn')

# Now, let's try a division by zero
array = np.array([1, 2, 3, 0])
result = 1 / array
print(result)
```

In this example, we're telling NumPy to ignore division by zero errors and just warn us about invalid operations. When we divide 1 by our array, instead of raising an error for the division by zero, it returns 'inf' (infinity) for that element.

■ The `geterr()` Function:

The `geterr()` function is like checking the current settings on our error control panel. It tells us how NumPy is currently set up to handle different types of errors.

```
import numpy as np

# Check current error settings
print(np.geterr())

# Change a setting
np.seterr(over='raise')

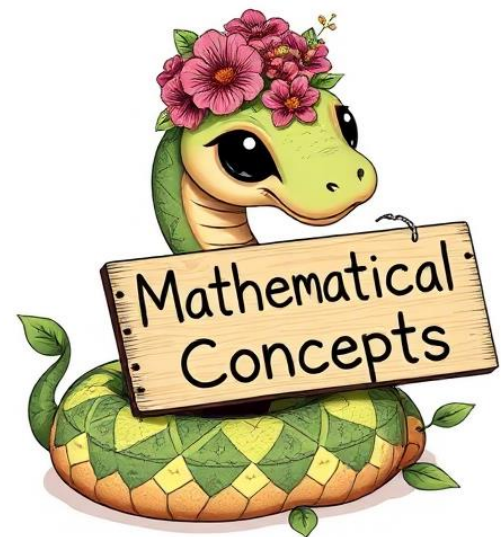
# Check again
print(np.geterr())
```

This is useful when you're debugging or when you want to ensure your error settings are correct before running a critical calculation.

Practical tips:

- Always check your error settings before running critical calculations.
- Use 'warn' instead of 'ignore' during development to catch potential issues.
- Remember that changing error settings affects the entire NumPy environment, so be cautious when working with shared code.
- For production code, consider using try/except blocks for more fine-grained error handling.

Remember, the goal isn't always to avoid errors entirely, but to handle them in a way that makes sense for your specific analysis.



Mathematical Concepts

Review of basic arithmetic operations

Let's start with the basics: adding, subtracting, multiplying, and dividing numbers. But here's the cool part - with NumPy, you can do these operations on entire arrays of numbers all at once!

Here's a quick tip to get started: sometimes you want to update your existing array instead of creating a new one. That's where in-place operations come in handy! Let's say you're updating elevation data with new measurements; you can simply use the `+=` operator for this. Let's have a look at an example:

```
elevation = np.array([100, 150, 200, 250])
new_data = np.array([5, -2, 3, 1])

elevation += new_data
print(elevation) # Output: [105 148 203 251]
```

The `+=` operator updates the 'elevation' array directly. This works just the same for subtraction (`-=`), multiplication (`*=`), and division (`/=`).

More universal functions

As we have seen before, unary ufuncs operate on a single input array. They're like solo performers, transforming each element of an array one by one. Let's look at some examples using elevation data:

```
import numpy as np

# Create an array of elevation data (in meters)
elevations = np.array([100, 150, 200, 250, 300])

# Convert elevations to feet
elevations_feet = np.round(np.multiply(elevations,
3.28084), 2)
print("Elevations in feet:", elevations_feet)

# Calculate the square root of elevations
elevation_sqrt = np.sqrt(elevations)
print("Square root of elevations:", elevation_sqrt)

# Find the absolute values
elevation_abs = np.abs(elevations - 200)
print("Absolute difference from 200m:", elevation_abs)
```

In this example, we used three unary ufuncs:

- `np.round()`: Rounds the values to 2 decimal places
- `np.sqrt()`: Calculates the square root of each element
- `np.abs()`: Computes the absolute value

These functions operate on each element of the array independently, making them perfect for element-wise transformations.

But we are not limited to operations on single arrays only. We can also have binary ufuncs, which are functions that work with two input arrays, performing operations between corresponding elements. Let's see some binary ufuncs in action with more geospatial data.

```
import numpy as np

# Create two arrays of latitude and longitude
coordinates
latitudes = np.array([40.7128, 34.0522, 41.8781,
37.7749, 39.9526])
longitudes = np.array([-74.0060, -118.2437, -87.6298, -
122.4194, -75.1652])

# Calculate the difference between latitudes and
longitudes
```

```
lat_diff = np.subtract(latitudes, np.mean(latitudes))
lon_diff = np.subtract(longitudes, np.mean(longitudes))
print("Latitude differences:", lat_diff)
print("Longitude differences:", lon_diff)

# Compute the Euclidean distance (simplified for
demonstration)
distances = np.sqrt(np.add(np.square(lat_diff),
np.square(lon_diff)))
print("Distances from mean point:", distances)

# Find the maximum values between latitudes and
longitudes
max_coords = np.maximum(latitudes, longitudes)
print("Maximum values:", max_coords)
```

Look familiar? In this example, we used several binary ufuncs:

- `np.subtract()`: Subtracts one array from another
- `np.add()`: Adds two arrays element-wise
- `np.square()`: Squares each element (technically unary, but used in a binary context)
- `np.maximum()`: Returns the element-wise maximum of two arrays

Remember, ufuncs are your friends when working with large geospatial datasets. They're fast, efficient, and designed to make your life easier.

Basic mathematical functions

In NumPy, we have two main functions for calculating absolute values: `np.abs()` and `np.fabs()`.

- `np.abs()`: This function works with both real and complex numbers.
- `np.fabs()`: This one is slightly faster but only works with real numbers (but has a fabulous name!).

```
import numpy as np

# Creating a sample array of elevations (in meters)
elevations = np.array([-10, 0, 15, -5, 20])

print("Original elevations:", elevations)
print("Absolute elevations (abs):", np.abs(elevations))
print("Absolute elevations (fabs):",
np.fabs(elevations))

# Output:
Original elevations: [-10  0 15 -5 20]
Absolute elevations (abs): [10  0 15  5 20]
Absolute elevations (fabs): [10.  0. 15.  5. 20.]
```

NumPy provides several rounding functions, each with its own behaviour:

- `np.round()`: Rounds to the nearest integer (or to a specified number of decimal places).
- `np.floor()`: Rounds down to the nearest integer.

- █ `np.ceil()`: Rounds up to the nearest integer.
- █ `np.trunc()`: Truncates the decimal part (rounds towards zero).

Let's see these in action with some latitude coordinates:

```
latitudes = np.array([37.7749, 40.7128, 34.0522,
41.8781, 29.7604])

print("Original latitudes:", latitudes)
print("Rounded:", np.round(latitudes, 2))
print("Floored:", np.floor(latitudes))
print("Ceiled:", np.ceil(latitudes))
print("Truncated:", np.trunc(latitudes))

# Output:
Original latitudes: [37.7749 40.7128 34.0522 41.8781
29.7604]
Rounded: [37.77 40.71 34.05 41.88 29.76]
Floored: [37. 40. 34. 41. 29.]
Ceiled: [38. 41. 35. 42. 30.]
Truncated: [37. 40. 34. 41. 29.]
```

These rounding functions can be useful when you need to adjust the precision of your data or when grouping data into bins.

The square root function, `np.sqrt()`, is incredibly useful in many geometric calculations (remember Pythagoras?):

```
# Calculating distances between two points
x1, y1 = 0, 0
x2, y2 = 3, 4

distance = np.sqrt((x2 - x1)**2 + (y2 - y1)**2)
print(f"Distance between (0,0) and (3,4): {distance}")

# Output: Distance between (0,0) and (3,4): 5.0
```

NumPy also provides several functions for exponents and logarithms:

- █ `np.exp()`: Calculates e^x
- █ `np.log()`: Natural logarithm (base e)
- █ `np.log10()`: Base-10 logarithm
- █ `np.log2()`: Base-2 logarithm

Remember that NumPy functions often work element-wise on arrays. This means you can apply these functions to entire arrays of data at once, which is incredibly powerful and efficient. But be careful with rounding errors in floating-point calculations. Sometimes, you might need to use `np.isclose()` instead of direct equality comparisons.

Trigonometric functions

We can, of course, also perform trigonometric functions with NumPy, whether they are basic,

inverse, hyperbolic, or inverse hyperbolic trigonometric functions.

Basic Trigonometric Functions

The three main trigonometric functions are sine (`sin`), cosine (`cos`), and tangent (`tan`). In NumPy, we can easily apply these functions to arrays of numbers.

```
import numpy as np

latitudes = np.array([0, 30, 45, 60, 90]) # in degrees

# Convert to radians (NumPy trig functions expect radians)
lat_radians = np.radians(latitudes)

sin_lat = np.sin(lat_radians)
cos_lat = np.cos(lat_radians)
tan_lat = np.tan(lat_radians)

print("Sine of latitudes:", sin_lat)
print("Cosine of latitudes:", cos_lat)
print("Tangent of latitudes:", tan_lat)
```

It's important to note that NumPy's trig functions work with radians, not degrees. That's why we use `np.radians()` to convert our latitudes first.

Inverse Trigonometric Functions

Now, what if we want to go the other way? That's where inverse trig functions come in. NumPy provides `arcsin`, `arccos`, and `arctan`. Let's say we have some sine values and want to find the corresponding angles:

```
sine_values = np.array([0, 0.5, 1])

arcsin_result = np.arcsin(sine_values)
arccos_result = np.arccos(sine_values)
arctan_result = np.arctan(sine_values)

print("Arcsine (in radians):", arcsin_result)
print("Arccosine (in radians):", arccos_result)
print("Arctangent (in radians):", arctan_result)

# Convert back to degrees
print("Arcsine (in degrees):",
np.degrees(arcsin_result))
```

Hyperbolic Functions

The fun doesn't stop here, though! Hyperbolic functions (`sinh`, `cosh`, `tanh`) might sound intimidating, but they're just extensions of the regular trig functions.

```
x = np.array([0, 1, 2, 3])

sinh_x = np.sinh(x)
cosh_x = np.cosh(x)
tanh_x = np.tanh(x)

print("Hyperbolic sine:", sinh_x)
print("Hyperbolic cosine:", cosh_x)
print("Hyperbolic tangent:", tanh_x)
```

Hyperbolic functions can be used in modelling certain types of curves in 3D space, which can be useful in terrain modelling or analysing satellite orbits.

Inverse Hyperbolic Functions

Just like with regular trig functions, we also have inverse hyperbolic functions: `arcsinh`, `arccosh`, and `arctanh`.

Statistical functions

NumPy offers a wide array of statistical functions that can help us make sense of large datasets. These functions allow us to uncover patterns, trends, and relationships in our data.

■ Mean, Median, and Mode

Let's start with some basic measures of central tendency: `np.mean()` and `np.median()`. NumPy does not have a built-in `mode()` function, but we can simply import `scipy` and use it that way.

```
elevations = np.array([100, 150, 200, 250, 300])

mean_elevation = np.mean(elevations)
print(f"Mean elevation: {mean_elevation} meters")

median_elevation = np.median(elevations)
print(f"Median elevation: {median_elevation} meters")
```

■ Standard Deviation and Variance

We can also look at measures about the spread of our data: standard deviation and variance. Standard deviations measure the amount of variation or dispersion in a set of values and can be calculated with the `np.std()` function.

```
rainfall = np.array([50, 60, 70, 80, 90, 100])
std_rainfall = np.std(rainfall)
print(f"Standard deviation of rainfall: {std_rainfall:.2f} mm")
```

You might remember that variance is the square of the standard deviation. We could either calculate it manually or use the `np.var()` function for vectorized operation.

■ Minimum and Maximum

The `min()` and `max()` functions help us find the extremes in our data and are pretty self-explanatory.

```
wind_speeds = np.array([5, 10, 15, 20, 25, 30])
min_speed = np.min(wind_speeds)
max_speed = np.max(wind_speeds)
print(f"Minimum wind speed: {min_speed} km/h")
print(f"Maximum wind speed: {max_speed} km/h")
```

■ Percentiles and Quantiles

The `np.percentile()` and `np.quantile()` functions help us understand the distribution of our data. `np.percentile()` gives us values below which a certain percentage of observations fall.

```
temperatures = np.array([20, 22, 24, 26, 28, 30, 32, 34, 36])
percentile_75 = np.percentile(temperatures, 75)
print(f"75th percentile of temperatures: {percentile_75} degrees Celsius")
```

On the other hand, `np.quantile()` is similar to percentiles, but expressed as fractions between 0 - 1.

```
quantile_50 = np.quantile(temperatures, 0.5)
print(f"50th quantile (median) of temperatures: {quantile_50} degrees Celsius")
```

Practical use case: These can help you understand the distribution of temperatures in a region. For example, the 75th percentile tells you the temperature that 75% of the observations fall below.

Linear algebra operations

NumPy further contains some powerful tools that will make your life much easier when dealing with matrices and vectors. Don't worry if some of these concepts seem intimidating at first – we'll break them down step by step.

Two operations that we've seen before are the dot product and matrix multiplication. The dot product is a fundamental operation in linear algebra. It's a way to multiply two arrays that can have different shapes. We have briefly seen the `np.dot()` function before when we looked at broadcasting.

```
# Let's say we have two vectors representing rainfall and temperature
rainfall = np.array([10, 20, 30])
temperature = np.array([25, 30, 35])
result = np.dot(rainfall, temperature)
print(f"Dot product: {result}")
```

Matrix multiplication is another crucial operation, for methods like transforming coordinates or applying filters to satellite imagery. NumPy provides two ways to do this: the `@` operator and the `np.matmul()` function. If you're unsure about the result of multiplying matrices with different shapes, have another look at the section on broadcasting.

```
elevation_1 = np.array([[100, 200], [300, 400]])
elevation_2 = np.array([[1, 2], [3, 4]])
result_1 = elevation_1 @ elevation_2
result_2 = np.matmul(elevation_1, elevation_2)
print(f"Using @ operator:\n{result_1}")
print(f"Using matmul:\n{result_2}")
```

The inner product, `np.inner()`, is similar to the dot product but more flexible with dimensions. The outer product, `np.outer()`, on the other hand, creates a matrix from two vectors. This is something that is commonly used in machine learning algorithms.

```
# Let's use our rainfall and temperature vectors again
inner_result = np.inner(rainfall, temperature)
outer_result = np.outer(rainfall, temperature)
print(f"Inner product: {inner_result}")
print(f"Outer product:\n{outer_result}")
```

The cross product, `np.cross()`, is primarily used with 3D vectors and can be helpful in calculating normal vectors to surfaces in 3D geospatial data. The cross product takes two 3D vectors and gives you a new vector that points in a direction perpendicular to both.

```
# Let's create two 3D vectors representing direction
vector_1 = np.array([1, 0, 0])
vector_2 = np.array([0, 1, 0])
cross_result = np.cross(vector_1, vector_2)
print(f"Cross product: {cross_result}")
```

Eigenvalues and eigenvectors are crucial in many machine learning algorithms, like principal component analysis and cluster analysis. NumPy offers the `np.linalg.eig()` function to calculate both. We'll see these much more later in the course.

```
# Using our transform_matrix from before
eigenvalues, eigenvectors =
np.linalg.eig(transform_matrix)
print(f"Eigenvalues: {eigenvalues}")
print(f"Eigenvectors:\n{eigenvectors}")
```

Solving linear systems is essential for operations like least squares fitting. NumPy makes light work of a quite math-heavy task too with the `np.linalg.solve()` function that solves any linear systems.

```
# Let's solve a system representing a coordinate
transformation
A = np.array([[2, 1], [1, 3]])
b = np.array([4, 5])
x = np.linalg.solve(A, b)
print(f"Solution: {x}")
```

And finally, inverting matrices is often used in transforming or rotating existing data. We use `np.linalg.inv()` for this.

Remember, these operations are the building blocks for more complex analyses. As you work with real-world data, you'll find yourself combining these operations in various ways.

Fourier transforms

In this section, we're going to explore some powerful tools that help us analyse and manipulate signals in the frequency domain. Fourier transforms are mathematical techniques that allow us to convert signals between the time domain (where we usually observe them) and the frequency domain (which reveals their underlying periodic components). This is incredibly useful in many fields, including image and signal processing, audio analysis, and even geospatial data analysis.

NumPy provides efficient implementations of various Fourier transform algorithms. We'll focus on four main functions: `fft`, `ifft`, `rfft`, and `fftfreq`.

Fast Fourier Transform (`fft`)

The Fast Fourier Transform, or FFT, is an algorithm that quickly computes the Discrete Fourier Transform of a signal. In NumPy, we use the `fft.fft()` function for this. The Fast Fourier Transform takes a signal, like a sound or a wave, and breaks it down into a bunch of simpler waves or frequencies, so you can see what it's made of, like finding out what musical notes make up a chord.

```
import numpy as np
import matplotlib.pyplot as plt
# Generate sample data (100 points of elevation data)
x = np.linspace(0, 10, 100)
elevation = 5 * np.sin(2 * np.pi * x) + 2 * np.sin(10 *
np.pi * x) + np.random.normal(0, 0.5, 100)
# Compute the FFT
fft_result = np.fft.fft(elevation)
# Plot the original signal and its frequency spectrum
plt.figure(figsize=(12, 6))
plt.subplot(211)
plt.plot(x, elevation)
plt.title('Original Elevation Data')
plt.subplot(212)
plt.plot(np.abs(fft_result))
plt.title('Frequency Spectrum')
plt.show()
```

This example generates synthetic elevation data with two sine waves and some noise, then computes its FFT. The resulting plot shows both the original signal and its frequency spectrum. We are using `matplotlib` here to plot the transform. We'll look at `matplotlib` in much greater depth in later chapter. For now, don't worry too much about the ins and outs of plotting data.

The FFT output is complex-valued. We often use `np.abs()` to get the magnitude spectrum, which tells us the strength of each frequency component.

■ Inverse Fast Fourier Transform (`ifft`)

The Inverse Fast Fourier Transform (IFFT) does the opposite of FFT – it converts frequency domain data back into the time domain. In NumPy, we use `fft.ifft()` for this. Let's continue with our elevation data example:

```
# Reconstruct the signal using IFFT
reconstructed = np.fft.ifft(fft_result)
# Plot the original and reconstructed signals
plt.figure(figsize=(12, 6))
plt.plot(x, elevation, label='Original')
plt.plot(x, reconstructed.real, '--',
label='Reconstructed')
plt.legend()
plt.title('Original vs Reconstructed Elevation Data')
plt.show()
```

This code reconstructs our original signal from its frequency components. Notice how closely it matches the original! IFFTs are often used in signal processing to remove noise or unwanted frequencies from data before converting it back to the time domain.

■ Real Fourier Transform (`rfft`)

For real-valued input (which is common in many applications), NumPy provides the `fft.rfft()` function. It's more efficient than the regular FFT for real data because it only computes the positive frequencies.

```
# Compute the real FFT
rfft_result = np.fft.rfft(elevation)
# Plot the original signal and its real frequency spectrum
plt.figure(figsize=(12, 6))
plt.subplot(211)
plt.plot(x, elevation)
plt.title('Original Elevation Data')
plt.subplot(212)
plt.plot(np.abs(rfft_result))
plt.title('Real Frequency Spectrum')
plt.show()
```

Notice how the output is about half the size of the full FFT. This is because `rfft` only returns the positive frequencies, which is all we need for real-valued input.

■ Fourier Frequencies (`fftfreq`)

When working with FFTs, it's often useful to know the frequencies corresponding to each point in the output. NumPy's `fft.fftfreq()` function helps us generate these frequencies.

Let's add frequency labels to our spectrum plot:

```
# Generate frequency values
freqs = np.fft.fftfreq(len(elevation), d=(x[1]-x[0]))
# Plot the frequency spectrum with labelled x-axis
plt.figure(figsize=(12, 6))
plt.plot(freqs[:len(freqs)//2],
np.abs(fft_result)[:len(freqs)//2])
plt.xlabel('Frequency (Hz)')
```

```
plt.ylabel('Magnitude')
plt.title('Frequency Spectrum with Labelled
Frequencies')
plt.show()
```

This plot now shows the actual frequencies present in our data, making it much easier to interpret! Remember that `fftfreq` returns both positive and negative frequencies. For most applications, we only need the positive half.

Fourier transforms are powerful tools, but they do have some limitations. They work best with periodic signals and assume that your data repeats infinitely and that time points are evenly-spaced. For non-periodic data or when analysing localized features, you might want to explore other techniques like wavelet analysis.

Random number generation and probability

In previous sections we generated random numbers using NumPy. Let's look at this in a little more detail. NumPy provides a powerful suite of tools for generating random numbers drawn from different distributions. These are incredibly useful for simulations, statistical modelling, and even creating test data. The key thing to remember is that computers can't generate truly random numbers – they use algorithms to produce numbers that appear random. That's why we call them "pseudo-random" numbers.

You'll remember that the `np.random.rand()` function gives you random numbers from a uniform distribution between 0 and 1. On the other hand, the `np.random.randn()` function gives you random numbers from a standard normal distribution (mean 0, standard deviation 1).

■ Setting Random Seeds

Sometimes, you want your random numbers to be reproducible. That's where seeds come in. By setting a seed, you ensure that the same sequence of random numbers is generated each time you run your code.

```
np.random.seed(42) # Set the seed to 42
print(np.random.rand(3))
# Will always output: [0.37454012 0.95071431 0.73199394]
np.random.seed(42) # Reset the seed
print(np.random.rand(3))
# Will output the same numbers: [0.37454012 0.95071431
0.73199394]
```

This is rather helpful when you're debugging or when you want to share your results with others.

Just be cautious when using random numbers for cryptographic purposes - NumPy's random number generators are not suitable for this.

■ Shuffling Arrays

Sometimes you want to randomize the order of elements in an array: `np.random.shuffle()`

```
arr = np.array([1, 2, 3, 4, 5])
np.random.shuffle(arr)
print(arr)
# Output might be: [3 1 5 4 2]
```

In addition to generating numbers from normal distributions with `np.random.normal()` or from uniform distributions with `np.random.uniform()`, NumPy provides functions to generate random numbers from poisson distributions too with `np.random.poisson()`.

```
# Generate 1000 points from a Poisson distribution with
mean 3
tree_counts = np.random.poisson(3, 1000)
```

Remember that `np.random.rand()` and `np.random.randn()` can take multiple arguments to generate multi-dimensional arrays. For example, `np.random.rand(3,3)` will give you a 3x3 array of random numbers.

Polynomial operations

NumPy offers a suite of functions for working with polynomials. These functions can be incredibly useful when you're analysing trends in your data, fitting curves, or even modelling complex processes. Here, we'll focus on three key operations: polynomial evaluation, finding roots, and fitting polynomials to data.

■ Polynomial Evaluation (`np.polyval()`)

Let's start with polynomial evaluation. Imagine you have a polynomial equation that describes, say, the elevation of a hiking trail as you move along its length. You can use the `np.polyval()` function to calculate the elevation at any given point.

`np.polyval()` takes two arguments: an array of polynomial coefficients (in descending order of degree) and the x-value(s) at which you want to evaluate the polynomial.

```
import numpy as np
# Define a polynomial: 2x^2 + 3x + 1
coefficients = [2, 3, 1]
# Evaluate the polynomial at x = 2
x = 2
result = np.polyval(coefficients, x)
```

```
print(f"The value of 2x^2 + 3x + 1 at x = 2 is:
{result}")
```

In this case, we're evaluating $2x^2 + 3x + 1$ at $x = 2$. The result should be $2(2^2) + 3(2) + 1 = 13$.

■ Finding Polynomial Roots (`np.roots()`)

In geospatial contexts, roots might represent points where a curve intersects with a baseline, like where a terrain profile crosses sea level.

`np.roots()` takes an array of polynomial coefficients (again, in descending order of degree) and returns the roots of the polynomial. Here's an example:

```
import numpy as np
# Define a polynomial: x^2 - 4
coefficients = [1, 0, -4]
# Find the roots
roots = np.roots(coefficients)
print(f"The roots of x^2 - 4 are: {roots}")
```

This should give us the roots -2 and 2, as these are the values of x where $x^2 - 4 = 0$. This is particularly useful when you are calculating maxima or minima in curves.

■ Fitting Polynomials (`np.polyfit()`)

Last but certainly not least, let's look at polynomial fitting. This is super useful when you have a set of data points and you want to find a polynomial that best describes the relationship between your x and y values.

`np.polyfit()` takes three main arguments: your x-values, your y-values, and the degree of the polynomial you want to fit. It returns the coefficients of the best-fit polynomial. Here's an example using some made-up elevation data:

```
import numpy as np
import matplotlib.pyplot as plt
# Sample data: distance along a trail (km) and elevation
(m)
distance = np.array([0, 1, 2, 3, 4, 5])
elevation = np.array([100, 125, 160, 210, 270, 350])
# Fit a 2nd degree polynomial
coefficients = np.polyfit(distance, elevation, 2)
# Generate points for a smooth curve
x_smooth = np.linspace(0, 5, 100)
y_smooth = np.polyval(coefficients, x_smooth)
# Plot the results
plt.scatter(distance, elevation, label='Data points')
plt.plot(x_smooth, y_smooth, 'r-', label='Fitted curve')
plt.xlabel('Distance (km)')
plt.ylabel('Elevation (m)')
plt.legend()
plt.title('Elevation Profile of a Trail')
plt.show()
print(f"The coefficients of the fitted polynomial are:
{coefficients}")
```

This script fits a quadratic curve to our elevation data and plots both the original points and the fitted curve.

Remember that `np.polyval()` and `np.roots()` use coefficients in descending order of degree (highest power first). When fitting polynomials, be cautious of overfitting. Higher degree polynomials might fit your data points perfectly but could be unrealistic and perform poorly on new data – much more on this in the machine learning section of this course.

Numerical integration

Let's explore two powerful methods for approximating definite integrals: the Trapezoidal Rule and Simpson's Rule. These techniques are incredibly useful when you're working with discrete data points and need to estimate the area under a curve.

In many real-world scenarios, we don't have a nice, neat function to integrate. Instead, we often have a set of data points. Numerical integration helps us estimate the area under the curve formed by these points. It's like trying to measure the area of an irregularly shaped field by approximating it with simpler shapes.

■ Trapezoidal Rule (`np.trapz`)

The Trapezoidal Rule is like laying a series of trapezoids under our curve and summing their areas. It's simple but effective, especially when you have evenly spaced data points. Here's how you can use it in NumPy:

```
import numpy as np
# Let's say we have elevation data (in meters) along a
# hiking trail

distances = np.array([0, 1, 2, 3, 4, 5]) # kilometers
along the trail
elevations = np.array([100, 150, 180, 220, 200, 180]) #
meters
area = np.trapz(elevations, distances)
print(f"The approximate area under the elevation curve
is {area} meter-kilometers")
```

In this example, we're estimating the area under an elevation profile of a hiking trail. The Trapezoidal Rule works best when your data points are closely spaced. If they're far apart, you might miss important features of the curve.

■ Simpson's Rule (`np.simps`)

Simpson's Rule is generally more accurate than the Trapezoidal Rule, especially for curves with changing slopes. It approximates the curve using parabolas instead of straight lines. Here's how to use Simpson's Rule in NumPy:

```
import numpy as np
from scipy import integrate
# Let's use rainfall data over an area
x_coords = np.linspace(0, 10, 11) # x coordinates in km
y_coords = np.linspace(0, 10, 11) # y coordinates in km
rainfall = np.array([
    [5, 7, 9, 12, 14, 16, 15, 13, 11, 8, 6],
    [6, 8, 11, 13, 15, 17, 16, 14, 12, 9, 7],
    [7, 9, 12, 14, 16, 18, 17, 15, 13, 10, 8],
    [8, 10, 13, 15, 17, 19, 18, 16, 14, 11, 9],
    [9, 11, 14, 16, 18, 20, 19, 17, 15, 12, 10],
    [10, 12, 15, 17, 19, 21, 20, 18, 16, 13, 11],
    [9, 11, 14, 16, 18, 20, 19, 17, 15, 12, 10],
    [8, 10, 13, 15, 17, 19, 18, 16, 14, 11, 9],
    [7, 9, 12, 14, 16, 18, 17, 15, 13, 10, 8],
    [6, 8, 11, 13, 15, 17, 16, 14, 12, 9, 7],
    [5, 7, 9, 12, 14, 16, 15, 13, 11, 8, 6]
]) # rainfall in mm
total_rainfall = integrate.simps([integrate.simps(row,
x_coords) for row in rainfall], y_coords)
print(f"The total rainfall over the area is
approximately {total_rainfall} square km-mm")
```

In this example, we're calculating the total rainfall over a 10km x 10km area. We first integrate along each row (x-direction) and then integrate the results in the y-direction. This gives us the volume of rainfall over the entire area. Cool, right?

Practical tips:

- Simpson's Rule generally provides more accurate results than the Trapezoidal Rule, especially for curves with changing slopes.
- Both methods assume that the data points are evenly spaced. If your data isn't evenly spaced, you'll need to use more complex versions of these algorithms.
- The accuracy of both methods improves with more data points. If you're dealing with sparse data, consider interpolating additional points before integration.
- For 2D data (like our rainfall example), you can apply these methods twice – once in each dimension.

Numerical differentiation

Numerical differentiation is a method used to approximate the derivative of a function using numerical methods. In simpler terms, it helps us understand how quickly a value is changing. This is incredibly useful in many fields, where we often need to understand rates of change.

■ Gradient Calculation (`np.gradient`)

Let's start with the gradient function. The gradient is a fancy word for the slope or rate of change in different directions. NumPy's gradient function calculates this for us across all dimensions of our data.

```
import numpy as np
# Let's create a 2D array representing elevation data
elevation = np.array([
    [100, 200, 300, 400],
    [150, 250, 350, 450],
    [200, 300, 400, 500]
])
# Calculate the gradient
gradient_y, gradient_x = np.gradient(elevation)
print("Gradient in Y direction:\n", gradient_y)
print("\nGradient in X direction:\n", gradient_x)
```

In this example, we're working with a 2D array that could represent elevation data. The gradient function returns two arrays: one for the vertical gradient (y-direction) and one for the horizontal gradient (x-direction). The gradient function works on any number of dimensions. For 1D data, it'll return a single array, for 3D data, it'll return three, and so on.

The values in these gradient arrays represent the rate of change at each point. A positive value means the elevation is increasing in that direction, while a negative value means it's decreasing.

■ Difference Calculations (`np.diff`)

While `np.gradient()` gives us a rate of change, `np.diff()` simply calculates the difference between adjacent elements.

```
import numpy as np
# Let's create a 1D array representing temperature
readings along a path
temperatures = np.array([20, 22, 25, 23, 21, 19])
# Calculate the differences
temp_diff = np.diff(temperatures)
print("Temperature differences:", temp_diff)
```

The `diff` function calculates the difference between each adjacent pair of values. This is great for identifying sudden changes or jumps in your data.

We can also use `diff` on 2D data:

```
# Using our elevation data from before
elevation_diff_y = np.diff(elevation, axis=0) # Differences along columns
elevation_diff_x = np.diff(elevation, axis=1) # Differences along rows
print("Elevation differences in Y direction:\n", elevation_diff_y)
print("\nElevation differences in X direction:\n", elevation_diff_x)
```

Here, we're calculating the differences along both axes of our 2D elevation data. The `np.diff()` function reduces the size of your array by 1 in the direction you're calculating differences. Keep this in mind when you're working with your data!

Complex number operations

Complex numbers are numbers with both a real and an imaginary component. In the real world, they're used in fields like signal processing, electromagnetic wave analysis, and even in some advanced geospatial calculations. We simply create complex numbers by using `np.complex(real, imaginary)`.

■ Real and Imaginary Parts

In NumPy, you can easily extract the real and imaginary parts of a complex number using the `real` and `imag` attributes.

```
# Let's create a complex number representing a 2D
coordinate
coordinate = np.complex(3, 4) # 3 + 4j
print("Real part:", coordinate.real) # Output: 3.0
print("Imaginary part:", coordinate.imag) # Output: 4.0
```

In this example, we've created a complex number that could represent a point on a 2D plane. The real part (3) might represent the x-coordinate, while the imaginary part (4) could represent the y-coordinate.

■ Complex Conjugate

The complex conjugate is useful in many calculations. It's the same number with the sign of the imaginary part flipped. NumPy has the `conj()` function for this.

```
# Using our coordinate from before
conjugate = np.conj(coordinate)
print("Original:", coordinate) # Output: (3+4j)
print("Conjugate:", conjugate) # Output: (3-4j)
```

The complex conjugate is often used to establish reflections and symmetry.

■ Angle and Absolute Value

The angle (also called the argument) and absolute value (or magnitude) of a complex number are crucial in many applications. Both are found using the `np.angle(complex number)` and `np.abs(complex number)` respectively.

```
# Calculate the angle (in radians) and absolute value
angle = np.angle(coordinate)
magnitude = np.abs(coordinate)
print("Angle (radians):", angle) # Output:
0.9272952180016122
print("Angle (degrees):", np.degrees(angle)) # Output:
53.13010235415598
print("Magnitude:", magnitude) # Output: 5.0
```

Let's say we're analysing the movement of a GPS-tracked vehicle. We can use complex numbers to represent its position and calculate its displacement:

```
# Initial position
start = np.complex(0, 0)
# Final position after some time
end = np.complex(3, 4)
# Calculate displacement
displacement = end - start
print("Displacement:", displacement) # Output: (3+4j)
print("Distance traveled:", np.abs(displacement)) #
Output: 5.0
print("Direction (degrees):",
np.degrees(np.angle(displacement))) # Output:
53.13010235415598
```

This tells us the vehicle moved 5 units in a direction of about 53.13 degrees from the positive x-axis. This technique might feel a little advanced but with a bit of practice it is highly useful when looking at displacement on maps.

Reduction operations

NumPy reduction operations are methods that reduce an array to a smaller size, often to a single value. They're called "reduction" operations because they reduce the dimensions of the data. These operations are super handy when you're dealing with large datasets and need to calculate totals, averages, or running totals.

■ Sum (`np.sum`)

The sum operation does exactly what you'd expect - it adds up all the elements in an array. It's like having a super-fast calculator that can add thousands of numbers in the blink of an eye!

```
import numpy as np
# Let's say we have elevation data for a mountain range
elevations = np.array([1000, 1500, 2000, 1800, 2200,
1600])
# To get the total elevation, we can use sum
total_elevation = np.sum(elevations)
print(f"Total elevation: {total_elevation} meters")
```

This would output: "Total elevation: 10100 meters"

■ Product (`np.prod()`)

The product operation, `np.prod()`, multiplies all the elements in an array. This can be really useful when you're dealing with growth rates or scaling factors. You could use this to calculate compound growth over time, like population growth or investment returns.

■ Cumulative Sum (`np.cumsum`)

The cumulative sum, `np.cumsum()`, gives you a running total. It's like keeping a running tally as you go through your data.

```
# Let's say we have monthly rainfall data
monthly_rainfall = np.array([50, 60, 70, 40, 30, 20])
# To get the cumulative rainfall, we use cumsum
cumulative_rainfall = np.cumsum(monthly_rainfall)
print("Cumulative rainfall:", cumulative_rainfall)
```

This would output: "Cumulative rainfall: [50 110 180 220 250 270]"

■ cumulative Product (`np.cumprod`)

The cumulative product, `np.cumprod()`, is similar to cumulative sum, but it multiplies as it goes along instead of adding.

All of these operations can work along different axes of multi-dimensional arrays. For example, `np.sum(array, axis=0)` sums along columns, while `np.sum(array, axis=1)` sums along rows.

Sliding window operations

Imagine you're looking at a landscape through a small, movable frame. As you slide this frame across the scene, you're performing a kind of sliding window operation. In NumPy, we use this concept to process arrays, applying operations to subsets of data as we move across the array.

Two key sliding window operations are convolution and correlation. Both are mathematical operations used to compare or combine two sequences of numbers (like lists or arrays). Convolution takes two sequences and produces a new sequence by flipping one of them, sliding it over the other, and multiplying their values together at each position, then adding up the results. Correlation is similar, but it doesn't flip the sequence—it just slides one over the other, multiplies the values at each position, and adds them up.

■ Convolution (`np.convolve`)

What is Convolution?

Convolution is like having a special lens that highlights certain features as you slide it across an image or dataset. It's widely used in image processing and signal processing

How it works:

- We have our main array (let's call it our "landscape").
- We also have a smaller array called a "kernel" or "filter".
- We slide this kernel over our landscape, multiply overlapping elements, and sum the results.

Let's see a simple example using elevation data:

```
import numpy as np
# Our landscape: a 1D array representing elevation along
# a path
landscape = np.array([10, 20, 30, 40, 30, 20, 10])
# Our kernel: a simple smoothing filter
kernel = np.array([0.3, 0.4, 0.3])
# Perform convolution
smoothed_landscape = np.convolve(landscape, kernel,
mode='valid')
print("Original landscape:", landscape)
print("Smoothed landscape:", smoothed_landscape)
```

In this example, we're smoothing out our elevation data. The kernel gives more weight to the central value and less to the neighbours, creating a gentle smoothing effect.

Convolution is fantastic for tasks like noise reduction in satellite imagery or smoothing out irregularities in terrain data.

■ Correlation (`np.correlate`)

Correlation measures how similar two arrays are as you slide one over the other. It's great for finding patterns or specific features in your data.

How it works:

- We have our main array (our "landscape" again).
- We have a "pattern" we're looking for.
- We slide the pattern over the landscape, multiply overlapping elements, and sum the results.

Let's look at an example using temperature data:

```
import numpy as np
# Our landscape: daily temperatures over a month
temperatures = np.array([20, 22, 25, 27, 28, 26, 24, 22,
21, 23, 25, 28, 30, 31, 29, 27])
# Pattern we're looking for: three days of rising
temperatures
pattern = np.array([1, 2, 3])
# Perform correlation
correlation = np.correlate(temperatures, pattern,
mode='valid')
print("Temperature data:", temperatures)
print("Correlation result:", correlation)
print("Potential heat wave starts at index:",
np.argmax(correlation))
```

In this example, we're looking for potential heat waves by correlating our temperature data with a pattern of rising temperatures.

When using `convolve` or `correlate`, be aware of how edge effects are handled. The `mode` parameter lets you control this. Use the `help()` function for more information on `mode`.

Often, it's helpful to normalize your data before applying these operations, especially when working with different scales.

Interpolation

Imagine you're a geologist studying the elevation of a mountain range. I really have to think of another example than elevation or long/lat data. You've collected data at specific points, but what about the areas in between? That's where interpolation comes in handy! Interpolation is like connecting the dots - it helps us estimate values between known data points.

In NumPy, we have some powerful tools for interpolation that can make our lives much easier. We'll focus on two main types: linear interpolation and polynomial interpolation.

■ Linear Interpolation (`np.interp`)

Let's start with linear interpolation. It's the simplest form of interpolation, assuming a straight line between two known points. Think of it as drawing a straight line between two elevation measurements on your mountain. Here's how you can use NumPy's `np.interp()` function for linear interpolation:

```
import numpy as np
# Known elevation points (in meters)
x = np.array([0, 5, 10, 15, 20]) # Distance along the
mountain (km)
y = np.array([100, 200, 150, 300, 250]) # Elevation (m)
# Points where we want to estimate elevation
x_new = np.array([2, 7, 13])
# Perform linear interpolation
y_new = np.interp(x_new, x, y)
print("Estimated elevations:", y_new)
```

In this example, we have known elevation points at specific distances along the mountain. We use `np.interp()` to estimate the elevation at new points (2 km, 7 km, and 13 km).

■ Polynomial Interpolation (`np.polyfit`)

Polynomial interpolation fits a polynomial curve to your data points, which can be more accurate for complex relationships. It's like drawing a smooth curve through your elevation points instead of straight lines:

```
import numpy as np
import matplotlib.pyplot as plt
# Known elevation points
x = np.array([0, 5, 10, 15, 20]) # Distance along the
mountain (km)
y = np.array([100, 200, 150, 300, 250]) # Elevation (m)
# Fit a 3rd degree polynomial
coeffs = np.polyfit(x, y, 3)
# Create a function from the coefficients
p = np.poly1d(coeffs)
# Generate points for a smooth curve
x_new = np.linspace(0, 20, 100)
y_new = p(x_new)
# Plot the results
plt.scatter(x, y, colour='red', label='Known points')
plt.plot(x_new, y_new, label='Interpolated curve')
plt.xlabel('Distance (km)')
```

```
plt.ylabel('Elevation (m)')
plt.legend()
plt.title('Mountain Elevation Profile')
plt.show()
```

In this example, we're fitting a 3rd-degree polynomial to our elevation data. The `np.polyfit()` function gives us the coefficients of this polynomial, which we then use to create a function with `np.poly1d()`. This function can estimate elevation at any point along our mountain range.

A word of warning: Higher degrees can lead to overfitting, especially with limited data points. Start with lower degrees (2 or 3) and increase only if necessary.

Integration with other scientific Python libraries

As we come to the end of the section on NumPy, where we looked at its fantastic applications and power, it is worth noting that it doesn't work in isolation – it plays nicely with other libraries to extend its capabilities even further.

■ NumPy and SciPy: A Perfect Pair

SciPy, or Scientific Python, builds on top of NumPy to provide additional functionality for scientific and technical computing. Think of NumPy as the foundation and SciPy as the advanced toolkit built on that foundation.

Let's see how we can use NumPy and SciPy together for some geospatial calculations:

```
import numpy as np
from scipy import spatial
# Let's say we have some latitude and longitude
coordinates
locations = np.array([
    [40.7128, -74.0060], # New York City
    [34.0522, -118.2437], # Los Angeles
    [41.8781, -87.6298], # Chicago
    [29.7604, -95.3698] # Houston
])
# We can use SciPy's spatial module to calculate
distances
distances = spatial.distance.pdist(locations)
print("Distances between cities (in degrees):")
print(distances)
```

Here, we're using NumPy to create an array of latitude and longitude coordinates, and then we're using SciPy's spatial module to calculate the distances between these points.

■ NumPy and SymPy: Bridging Numerical and Symbolic Math

While NumPy is great for numerical computations, sometimes we need to work with symbolic mathematics. That's where SymPy comes in. SymPy is a library for symbolic mathematics, and it can work seamlessly with NumPy.

Here's an example of how we might use NumPy and SymPy together in a geospatial context:

```
import numpy as np
import sympy as sp
# Define symbolic variables
x, y = sp.symbols('x y')
# Define a symbolic function (let's say it's a
simplified elevation model)
elevation = 100 * sp.sin(x/1000) * sp.cos(y/1000)
# Create a NumPy function from the SymPy expression
elevation_func = sp.lambdify((x, y), elevation, "numpy")
# Now we can use this with NumPy arrays
x_coords = np.linspace(0, 1000, 10)
y_coords = np.linspace(0, 1000, 10)
X, Y = np.meshgrid(x_coords, y_coords)
Z = elevation_func(X, Y)
print("Elevation values:")
print(Z)
```

After using SymPy to define a symbolic function representing a simplified elevation model, we convert this symbolic function into a NumPy-compatible function using `sp.lambdify()`. This allows us to apply the function to NumPy arrays, which is much faster than evaluating the symbolic expression for each point.

This combination is powerful because it allows us to define complex mathematical models symbolically (which is often easier to understand and manipulate) and then apply them efficiently to large datasets using NumPy.

And that's it for our section on NumPy. Hopefully you will have appreciated its raw power and ability to work with other libraries and seamlessly integrate with your Python program's workflow. NumPy is a big package and we have only scratched the surface looking at the most important features. It is important that you practice what you have learned to cement the knowledge and that you dive into NumPy's online document. Happy coding!