Working with Python

Marcus Kazmierczak

Version v1.0, 2025-06-13

Table of Contents

The Zen of Python

By Tim Peters

> Beautiful is better than ugly.
> Explicit is better than implicit.
> Simple is better than complex.
> Complex is better than complicated.
> Flat is better than nested.
> Sparse is better than dense.
> Readability counts.
> Special cases aren't special enough to break the rules.
> Although practicality beats purity.
> Errors should never pass silently.
> Unless explicitly silenced.
> In the face of ambiguity, refuse the temptation to guess.
> There should be one—and preferably only one—obvious way to do it.
> Although that way may not be obvious at first unless you're Dutch.
> Now is better than never.
> Although never is often better than **right** now.
> If the implementation is hard to explain, it's a bad idea.
> If the implementation is easy to explain, it may be a good idea.
> Namespaces are one honking great idea — let's do more of those!

Chapter 1. Introduction

Working with Python is a book focused on practical programming with Python. The intended audience is someone who is new to Python, but not necessarily new to programming. They want to learn how to get things done using Python. As someone who learns best through code examples, I've filled this guide with practical examples. What started as my personal reference guide has evolved into a full book that I still use regularly, and will keep evolving.

1.1. Support the Author

If you find *Working with Python* helpful, consider supporting it with a small donation. I put a lot of time and effort in putting this all together, pay what you think it is worth.

- Buy me a coffee
- Donate via PayPal

1.2. Setup

This guide assumes you have a working Python development environment, running **Python 3.10** or later. See the Dev Environment appendix to install and set up Python on your machine.

1.3. About the guide

The guide is organized into chapters, with the first nine chapters covering the basics of Python. The remaining chapters cover more advanced topics. If you are familiar with Python, starting on Python Idioms is a good place to start. Feel free to jump around to whatever section may interest you; it is intended as reference, not a linear guide.

I highly recommend trying and playing with the examples, **learning is an active process**. You will not gain much by reading through the examples. Open up the Python REPL and type them in and play with them, alter the examples, mess around and find out. It is the best way to learn.

1.3.1. Code Examples

The Python code examples can be pasted directly into the REPL. To support this, I reverse the `>>>` that indicate an input line in the REPL, and switch it with the output line, which has no prefix.

In your terminal, the join list example would look like:

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> a + b
[1, 2, 3, 4, 5, 6]
```

But copying and pasteing that is tricky, especially for multiple lines. So I reverse the delimiter, making it easier to copy & paste examples, though I recommend typing them in yourself. It will help you learn.

In the guide, the example is shown as:

```
a = [1, 2, 3]
b = [4, 5, 6]
a + b
>>> [1, 2, 3, 4, 5, 6]
```

Additionally, the REPL automatically displays the return value or variable as shown above. So, I tend not to use `print()`, unless I am showing a specific example. It is not necessary to have `print(a + b)` or an even more verbose `c = a + b` and then another line to `print(c)`.

One issue, unfortunately, PDF support for tabs/spaces is not great, so watch for proper indentation whenever you copy-paste blocks of code, another good reason to type the code yourself.

1.4. License

Chapter 2. Numbers

Start off with the basics, working with numbers.

2.1. Convert string to integer

To use a string as an integer, you must explicitly convert it. Use the `int()` function for this conversion. You will get a `TypeError` if you try to use a string as an integer without converting it.

```
s = '13'
s + 2
>>> TypeError: can only concatenate str (not "int") to str
```

Instead use:

```
s = '13'
int(s) + 2
>>> 15
```

Be aware that the `+` operator is also used for joining strings and lists together, so you might get strange results if you don't convert first.

```
s = '13'
t = '2'
s + t
>>> '132'
```

2.2. Convert string to float

Use `float()` to convert a string to a float, or to convert an integer to a float.

```
n = 23
float(n)
>>> 23.0
```

2.3. Check if string is a number

Python doesn't have a simple built-in way to check if a string is a number. You can use `.isdigit()` for positive integers and `.isnumeric()` for Unicode numeric characters, but they will fail on negative numbers and decimals. This function is often your best option for a general check:

```
def is_number(s):
    try:
```

```
        float(s)
        return True
    except ValueError:
        return False
```

2.4. Modulo Arithmetic

To perform modulo arithmetic in Python, use `//` to get the floored quotient (the whole number in division) and `%` to get the remainder. For example:

```
13 // 4
>>> 3

13 % 4
>>> 1
```

2.5. Exponent Arithmetic

There are two ways to calculate powers: use the `**` operator or the `pow()` function. They will give you the same results:

```
4 ** 2
>>> 16

pow(4, 2)
>>> 16
```

2.6. Absolute Value

Use the `abs()` function to get the absolute value of a number:

```
abs(-3)
>>> 3
```

2.7. Sum Numbers in a List

Python has a built-in `sum()` function to sum elements in a list or other iterable.

```
my_numbers = [1, 2, 3]
sum(my_numbers)
>>> 6
```

2.8. Underscores for Long Numbers

You can use underscores in long numbers to make them easier to read. The Python interpreter ignores them.

```
x = 100000000
y = 100_000_000   # like commas
z = 10_00_0_0000  # weird, but valid
x == y == z
>>> True
```

If you want to format a number with commas for display, use the `{:,}` f-string format. See String Formatting for more examples on formatting.

```
y = 100_000_000
print(f"{y:,}")
>>> 100,000,000
```

Chapter 3. Strings

A set of examples for working with strings in Python.

3.1. Defining Strings (Delimiters)

You can use single or double quotes to specify a string in Python. This doesn't change the interpretation of the string, unlike in some other languages like PHP.

```
s1 = 'Single quote'
s2 = "Double quote"
```

Special characters, such as `\n` for a new line, will be converted:

```
print('String with a \n new line')

>>> String with a
>>>  new line
```

You can also use triple quotes (either single or double) to create strings that span multiple lines:

```
print("""This string
spans multiple
lines""")

>>> This string
>>> spans multiple
>>> lines
```

3.1.1. Raw Strings

If you want a raw string (where backslashes are treated as literal characters and not escape sequences), prefix the string with an `r`:

```
print(r"This is a raw \n string")
>>> This is a raw \n string
```

3.2. Concatenating Strings

Python is flexible when it comes to joining strings. You can use the `+` operator:

```
s1 = "Hello"
s2 = "World"
```

```
s1 + s2
>>> 'HelloWorld'
```

Use the * operator to repeat a string:

```
s1 = "Repeat"
s1 * 3
>>> 'RepeatRepeatRepeat'
```

3.2.1. Joining a List of Strings

To join a list of strings into a single string, use the `join()` method. You call this method on the separator string you want to use between the elements.

```
letters = ['a', 'b', 'c']
'-'.join(letters)
>>> 'a-b-c'
```

3.3. Getting the Length of a String

Use the `len()` function to get the length of a string (this function also works for lists and other collections).

```
s = "Test String"
len(s)
>>> 11
```

3.4. Checking for an Empty String

You can use `len()` to check if a string is empty (i.e., has a length of 0). A more Pythonic way is to use the string itself in a boolean context; an empty string evaluates to `False`.

```
a = ""
if not a:
    print("String is empty")

>>> String is empty

b = "Hello"
if b:
    print("String is not empty")

>>> String is not empty
```

3.5. Checking if a Substring Exists

Python doesn't have a dedicated `contains()` method for strings. Instead, use the `in` operator to check if a string contains a substring:

```
s = "The dog has brown spots"
if "brown" in s:
    print("Yes, 'brown' is in the string")

>>> Yes, 'brown' is in the string
```

Use `not in` to check if a string does **not** contain a substring:

```
s = "The dog has brown spots"
if "red" not in s:
    print("No, 'red' is not in the string")

>>> No, 'red' is not in the string
```

Alternatively, use the `.find()` method, which returns the starting index of the substring if found, or -1 if not found. For a simple boolean check, `in` is often preferred for its clarity and consistency with list membership testing.

```
s = "The dog has brown spots"
idx = s.find("brown")
if idx != -1:
    print(f"Found 'brown' at position: {idx}")

>>> Found 'brown' at position: 12

idx_not_found = s.find("red")
if idx_not_found == -1:
    print("'red' was not found")

>>> 'red' was not found
```

3.6. String Methods

The Python string type has many useful built-in methods. Find the full documentation at stdtype string methods.

Remember: To use these methods, call them on a string instance, like `"this is my string".method()` or `my_string_variable.method()`.

3.6.1. Splitting Strings

To split a string into a list of substrings based on a specific delimiter, use the `split()` method.

```
s = "a-b-c"
s.split("-")
>>> ['a', 'b', 'c']
```

3.6.2. Replacing Substrings

To replace occurrences of a substring with another string, use `replace("old", "new")`:

```
s = "The quick brown fox jumped over the lazy dog"
s.replace("fox", "cat")
>>> 'The quick brown cat jumped over the lazy dog'
```

3.6.3. Stripping Characters

To remove characters from the beginning and end of a string, use `.strip()`. If you don't specify any characters, it removes whitespace (including spaces, tabs, and newlines) by default.

```
' This is my string \n'.strip()
>>> 'This is my string'
```

To specify your own set of characters to strip, pass them as an argument to `.strip()` (e.g., `.strip("rs")` will remove any 'r' or 's' characters from the ends).

```
"srabcdefsrs".strip("rs")
>>> 'abcdef'
```

Python also provides `.lstrip()` to strip characters only from the left (beginning) and `.rstrip()` to strip only from the right (end).

3.6.4. Checking for Prefixes or Suffixes

To determine if a string starts with a certain prefix or ends with a certain suffix, use the `.startswith()` or `.endswith()` methods respectively.

```
s = "prevention"
s.startswith("pre")
>>> True
s.endswith("tion")
>>> True
s.startswith("tion")
```

```
>>> False
```

Chapter 4. String Formatting Cookbook

Python 3.6 introduced formatted string literals, referred to as f-strings, as another method to help format strings. F-strings are a powerful and flexible way to format strings in Python. An f-string is a string prefixed with `f` or `F`. They allow you to embed expressions inside string literals, using curly braces `{}`. The expressions are evaluated at runtime and then formatted according to the format specification inside the braces.

F-strings are now the defacto standard way to format strings since they are far simpler to use and read. Using f-strings in Python is similar to JavaScript's template literals, if you are familiar with them.

Here's an example comparing the three ways to format a float number:

```
pi = 3.14159
print("pi: %1.2f" % pi)          # Older %-formatting
print("pi: {:.2f}".format(pi))   # .format() method
print(f"pi: {pi:.2f}")           # f-string

>>> pi: 3.14
>>> pi: 3.14
>>> pi: 3.14
```

This formatting guide will use f-strings as the primary method for formatting strings, though there are some benefits at the end to still use the `.format()` method.

4.1. Number formatting

This table shows various ways to format numbers using Python's formatted string literals, including examples for both float formatting and integer formatting.

To run examples use: `print(f"Number: {n:FORMAT}")`

Number	Format	Output	Description
3.1415926	`{n:.2f}`	3.14	Format float 2 decimal places
3.1415926	`{n:+.2f}`	+3.14	Format float 2 decimal places with sign
-1	`{n:+.2f}`	-1.00	Format float 2 decimal places with sign
2.71828	`{n:.0f}`	3	Format float with no decimal places
5	`{n:0>2d}`	05	Pad number with zeros (left padding, width 2)
5	`{n:x<4d}`	5xxx	Pad number with x's (right padding, width 4)
1000000	`{n:,}`	1,000,000	Number format with comma separator
0.25	`{n:.2%}`	25.00%	Format percentage
1000000000	`{n:.2e}`	1.00e+09	Exponent notation
13	`{:10d}`	13	Right aligned (default, width 10)
13	`{n:<10d}`	13	Left aligned (width 10)

Number	Format	Output	Description
13	`{n:^10d}`	13	Center aligned (width 10)

4.2. String formatting basics

Here are a couple of examples of basic string substitution

```
val = "money"
s1 = f"show me the {val}"

adj = "tasty"
noun = "burger"
s2 = f"hmmm, this is a {adj} {noun}"
```

4.2.1. Variable formatting

You can use variables inside the formatting brackets. For example using a precision variable to control how many decimal places to show:

```
pi = 3.1415926
precision = 4
print(f"pi to {precision} decimal places = {pi:.{precision}f}")
>>> 'pi to 4 decimal places = 3.1416'
```

4.2.2. Convert values to different bases

A surprising use with the string format command is to convert numbers to different bases. Use the letter in the formatter to indicate the number base: **d**ecimal, he**x**, **o**ctal, or **b**inary.

This example formats the number `21` in each base:

```
x = 21
f"{x:d} - {x:x} - {x:o} - {x:b}"
>>> '21 - 15 - 25 - 10101 '
```

4.3. Formatting Expressions

F-strings also support expressions and functions inside of the brackets `{ }` this allows you to:

4.3.1. Do math with f-strings:

```
print(f"Do math: 3 * 6 = {3 * 6}")
>>> Do math: 3 * 6 = 18
```

4.3.2. Call functions with f-strings;

```
verb = "runs"
print(f"The girl {verb.upper()} quickly.")
>>> The girl RUNS quickly.
```

4.3.3. Delimiting f-strings

Use f-strings with the three different type of quotation marks in Python, single, double, or triple quotes. The following will all output the same:

```
name = "Fred"
print(f'{name}')
print(f"{name}")
print(f"""{name}""")
```

4.3.4. F-String error

The one thing to be careful with is mixing the two formats, if `{}` are used inside of an f-string, it will give the error:

```
SyntaxError: f-string: empty expression not allowed
```

Each set of brackets used in an f-string requires a value or variable.

4.3.5. Internationalization

To use locale specific formatting for numbers, first set the locale, and then use the formating code `n` instead of `d`. For example, using commas or periods to separate thousands in numbers based on the user's locale.

Here is an example, setting locale and formatting a number to display the proper separator:

```
import locale
locale.setlocale(locale.LC_ALL, '')

print(f"{1000000:n}")
```

4.3.6. Escaping braces

If you need to use braces when using f-strings just double them up:

```
print(f"The empty set is often represented as {{0}}.")
>>> The empty set is often represented as {0}.
```

4.4. Use `.format` as a function

The `format()` function does allow for some additional features and capabilities, here are a couple of examples using the `.format` method to return a function that can be used to separate text and formatting from code. For example, at the beginning of a program include all the formats for later use.

```
## defining formats
email_f = "Your email address is {email}".format

## use elsewhere
print(email_f(email="bob@example.com"))
```

Using format as a function can be used to adjust formating by user preference.

```
## set user preferred format
num_format = "{:,}".format

## use elsewhere
print(num_format(1000000))
```

4.4.1. Table formatting data

Use the width and the left and right justification to align your data into a nice table format. Here's an example using the `.format` method with rows of data:

```
# data
starters = [
    [ 'Andre Iguodala', 4, 3, 7 ],
    [ 'Klay Thompson', 5, 0, 21 ],
    [ 'Stephen Curry', 5, 8, 36 ],
    [ 'Draymon Green', 9, 4, 11 ],
    [ 'Andrew Bogut', 3, 0, 2 ],
]

# define format row
row = "| {0:<16s} | {1:2d} | {2:2d} | {3:2d} |".format

for p in starters:
    print(row(*p))
```

Note: `*p` expands the tuple into individual arguments for the format function. See Functions chapter for more.

This would output:

```
| Andre Iguodala   |  4 |  3 |  7 |
| Klay Thompson    |  5 |  0 | 21 |
| Stephen Curry    |  5 |  8 | 36 |
| Draymon Green    |  9 |  4 | 11 |
| Andrew Bogut     |  3 |  0 |  2 |
```

Chapter 5. Lists

Lists are Python's primary built-in data structure for collections. They are **ordered**, and **mutable** (meaning you can change them); and a single list can contain items of different types.

5.1. Creating Lists

You define Python lists using square brackets `[ ]`.

```
my_list = ["a", "b", "c"]
```

5.2. Adding Items to a List

To add an item to the end of a list, use the `append()` method.

```
my_list = [1, 2, 3]
my_list.append(4)
my_list
>>> [1, 2, 3, 4]
```

To add an item to a specific position in a list, use the `insert(pos, el)` method, where `pos` is the index of the position to insert the item at, and `el` is the item to insert.

```
my_list = ["a", "b", "c"]
my_list.append("d")
my_list.insert(1, "x")
```

👉 What do you think the list will look like now? Try to predict before running the code. Remember the lists are indexed starting at 0.

5.3. Concatenating Two Lists

To concatenate two lists in Python, use the `+` operator.

```
my_list = [1, 2, 3]
another_list = [4, 5, 6]
my_list + another_list
>>> [1, 2, 3, 4, 5, 6]
```

5.4. Accessing Elements in a List

To access an element in a list, use square brackets `[ ]` with the index of the element.

```
my_list = ["a", "b", "c"]
my_list[0]
>>> "a"
```

You can also use negative indices to access elements from the end of the list:

```
my_list = ["a", "b", "c"]
my_list[-1]
>>> "c"
```

5.5. List Slicing

Use slicing to access a range of elements:

```
my_list = ["a", "b", "c", "d", "e"]
my_list[1:3]
>>> ["b", "c"]
```

If your slice omits the beginning index, it starts from the beginning of the list:

```
my_list = ["a", "b", "c", "d", "e"]
my_list[:3]
>>> ["a", "b", "c"]
```

If your slice omits the ending index, it goes until the end of the list:

```
my_list = ["a", "b", "c", "d", "e"]
my_list[2:]
>>> ["c", "d", "e"]
```

5.6. Counting Elements in a List

Use the `len()` function to get a count of all elements in a list.

```
my_list = ["a", "b", "c"]
len(my_list)
>>> 3
```

5.6.1. Counting Specific Elements

Use the `.count()` method to count the occurrences of a specific item in a list:

```
my_list = ["a", "b", "b", "c"]
my_list.count("b")
>>> 2
```

5.7. Initializing a Two-Dimensional List (Matrix)

To create a two-dimensional list (like a matrix) and initialize it, for example, with all zeros, you can use a list comprehension:

```
width, height = 3, 2 # Example dimensions
matrix = [[0 for x in range(width)] for y in range(height)]

for row in matrix:
    print(row)

>>> [0, 0, 0]
>>> [0, 0, 0]
```

5.8. Iterating Over a List

To iterate over the items in a list, you can use a `for` loop:

```
my_list = ["a", "b", "c"]
for item in my_list:
    print(item)

>>> a
>>> b
>>> c
```

5.8.1. Iterating Over a List with an Index

If you need both the index and the item while iterating, use the `enumerate()` function:

```
my_list = ["a", "b", "c"]
for idx, item in enumerate(my_list):
    print(f"{idx} -> {item}")

>>> 0 -> a
>>> 1 -> b
```

```
>>> 2 -> c
```

5.9. Sorting Lists

Python provides built-in ways to sort lists.

5.9.1. Sorting a List

Use the list's `.sort()` method to sort it in-place (this modifies the original list). By default, it sorts by value.

```
my_list = [3, 2, 1]
my_list.sort()
print(my_list)
>>> [1, 2, 3]


my_list = ["c", "b", "a"]
my_list.sort()
print(my_list)
>>> ['a', 'b', 'c']
```

Use the `sorted()` built-in function to return a new sorted list without modifying the original list.

```
my_list = [3, 2, 1]
for el in sorted(my_list):
    print(el)

>>> 1
>>> 2
>>> 3
```

5.9.2. Reversing a List

Use the list's `.reverse()` method to reverse its order in-place (this modifies the original list).

```
my_list = ["a", "b", "c"]
my_list.reverse()
print(my_list)
>>> ['c', 'b', 'a']
```

Similarly, use `reversed()` function to create a reversed iterator without modifying the original list.

```
my_list = ["a", "b", "c"]
```

```
for item in reversed(my_list):
    print(item)

>>> c
>>> b
>>> a
```

5.10. Removing Items from a List

Use the `.pop()` method to remove an item at a specific index. If you don't specify an index, it removes and returns the last item. `.pop()` returns the removed item.

```
my_list =["a", "b", "c"]
item = my_list.pop(1) # Removes item at index 1 ('b')
print(f"List: {my_list}, Item: {item}")

>>> List: ['a', 'c'], Item: b
```

5.10.1. Removing Items from a List by a Slice

Use the `del` statement to remove items by a slice.

```
my_list = ["a", "b", "c", "d", "e"]
del my_list[1:3] # Removes items at indices 1 and 2 ('b' and 'c')
print(my_list)
>>> ['a', 'd', 'e']
```

5.10.2. Removing an Item from a List by Value

You can remove an item by its value using the `.remove()` method. Be aware that this method only removes the **first** occurrence of the value and raises a `ValueError` if the value is not found.

```
my_list = ["a", "b", "b", "c"]
my_list.remove("b") # Removes the first occurrence of 'b'
print(my_list)
>>> ['a', 'b', 'c']

try:
    a.remove("z")
except ValueError as e:
    print(e)

>>> list.remove(x): x not in list
```

If you need to remove **all** items matching a certain value or condition, use a list comprehension or the `filter()` function. Note: `filter()` returns a filter object, so you'll need to convert it back to a list.

```
my_list = ["a", "b", "b", "c"]
# Using filter
filtered_list = list(filter(lambda el: el != "b", my_list))
filtered_list
>>> ['a', 'c']

# Using list comprehension (often more readable)
my_list = ["a", "b", "b", "c"]
comprehended_list = [el for el in my_list if el != "b"]
comprehended_list
>>> ['a', 'c']
```

Chapter 6. Dictionaries

A Python `dict` is an associative array, also known as a key-value store.

6.1. Creating dicts

Define a dictionary much like a JSON object definition.

```
d = {'k1': 'v1', 'k2': 'v2', 'k3': 'v3' }
```

Alternatively, use the `dict()` constructor to build a dictionary from a list of key-value tuples.

```
d = dict([ ('k1', 'v1'), ('k2', 'v2'), ('k3', 'v3') ])
```

6.2. Key Exists in Dictionary

Use the `in` keyword to check if a key exists in a dictionary.

```
d = {'k1': 'v1', 'k2': 'v2', 'k3': 'v3' }
if 'k1' in d:
    print("Key exists")
>>> Key exists
```

6.3. Iterating Through Dictionaries

6.3.1. Iterate over keys of a dict

Use the `.keys()` method to get an iterable of all keys in a dictionary. You can loop through this view directly.

```
d = {'k1': 'v1', 'k2': 'v2', 'k3': 'v3' }
for key in d.keys():
    print(key)
>>> k1
>>> k2
>>> k3
```

6.3.2. Iterate over values of a dict

Similarly, use the `.values()` method to get an iterable view of all values.

```
d = {'k1': 'v1', 'k2': 'v2', 'k3': 'v3' }
for val in d.values():
```

```
    print(val)
>>> v1
>>> v2
>>> v3
```

6.3.3. Iterate over dict using key, value pairs

To iterate over key-value pairs, use the `.items()` method. This gives you an iterable view of (key, value) tuples.

```
d = {'k1': 'v1', 'k2': 'v2', 'k3': 'v3' }
for k, v in d.items():
    print(k, v)
>>> k1 v1
>>> k2 v2
>>> k3 v3
```

6.4. Get item from dict

Retrieve an item using its key with the `.get()` method. Provide a second argument as a default value if the key might not exist, preventing a `KeyError`.

```
d = {'k1': 'v1', 'k2': 'v2', 'k3': 'v3' }
d.get('k1')
>>> v1

d.get('k4', 0)
>>> 0
```

6.5. Remove item from dict

Remove an item from a dictionary using the `del` statement followed by the dictionary and key. If the key doesn't exist, this will raise a `KeyError`.

```
d = {'k1': 'v1', 'k2': 'v2', 'k3': 'v3' }
del d['k2']
print(d) # To see the result
>>> {'k1': 'v1', 'k3': 'v3'}
```

6.6. Merge two dictionaries

Starting with Python 3.9, merge two dictionaries using the `|` (pipe) operator. If keys overlap, the values from the second (right-hand) dictionary will take precedence.

```
a1 = { 'a': 'apple', 'b': 'banana' }
a2 = { 'b': 'berry', 'c': 'cherry' }
a1 | a2
>>> { 'a': 'apple', 'b': 'berry', 'c': 'cherry' }
```

This is particularly useful for setting default configurations and overriding them with user-specific preferences.

6.7. Dictionary Order Preservation

Starting with Python 3.7, standard Python dictionaries remember the order in which items were inserted. When you iterate over a dictionary, items are retrieved in the same order they were added. This is often all you need for ordered behavior.

For versions of Python prior to 3.7, or if you need specialized features like reordering keys after creation or ensuring equality checks also consider order, use the `OrderedDict` from the `collections` module.

```
from collections import OrderedDict

od = OrderedDict()
od['a'] = "alpha"
od['b'] = "beta"
od['c'] = "gamma"

for k, v in od.items():
    print(f" {k} => {v}")
>>>  a => alpha
>>>  b => beta
>>>  c => gamma
```

6.7.1. Iterating in a Custom Sorted Order

While dictionaries preserve insertion order (Python 3.7+), you might need to iterate in a different order, such as alphabetically by key. To do this, retrieve the keys, sort them, and then iterate through the sorted list to access dictionary items.

```
d = {
    'b': 'beta',
    'a': 'alpha',
    'g': 'gamma',
}

keys = list(d.keys()) # Get keys and convert to a list
keys.sort()           # Sort the list of keys
for k in keys:
    print(f" {k} -> {d[k]}")
```

```
>>>  a -> alpha
>>>  b -> beta
>>>  g -> gamma
```

6.8. Get Dict Key by Max Value

To find the key corresponding to the maximum value in a dictionary, use the `max()` function. Set the `key` argument of `max()` to the dictionary's `.get` method.

```
scores = {
    'Alice': 78,
    'Biren': 64,
    'Charlie': 92,
    'David': 54,
    'Eva': 87
}

max_key = max(scores, key=scores.get)
print(max_key)
>>> Charlie
```

Chapter 7. Control Flow

7.1. Conditionals

7.1.1. Basic if conditionals

The basic `if` syntax in Python, with `elif` (else if) and `else`:

```
x = 5
if x > 0:
    print(f"{x} is positive")
elif x < 0:
    print(f"{x} is negative")
else:
    print("Zero. Your number is zero.")
```

7.1.2. Ternary operator

Python doesn't have a specific ternary operator. Instead, you can use `if-else` inline with the form `value_if_true if condition else value_if_false`. Note: the `else` portion is required.

```
x = 5
# Assign 'negative' if x < 0, otherwise 'non-negative'
sign_description = "negative" if x < 0 else "non-negative"
print(f"The number {x} is {sign_description}.")
# >>> The number 5 is non-negative.

x = -2
sign_description = "negative" if x < 0 else "non-negative"
print(f"The number {x} is {sign_description}.")
# >>> The number -2 is negative.
```

7.1.3. Match statement

The `match` statement, introduced in Python 3.10, allows you to match a value against a series of patterns in `case` blocks.

```
num = 3
match num:
    case 1:
        print("One - Not prime by definition")
    case 2 | 3 | 5 | 7:
        print("Prime number")
    case 4 | 6 | 8 | 9:
        print("Not prime")
```

```
    case _:
        print("Shrug")
```

The `case _` is a wildcard and will match anything not yet matched if all other cases fail.

Multiple values can be included in a `case` by separating them with the `|` operator (representing "or").

You aren't limited to matching simple values; you can match more complex objects too.

```
throw = ("r", "p")
# Rock-paper-scissor tuple
match throw:
    case ("r", "r") | ("p", "p") | ("s", "s"):
        print("Tie")
    case ("r", "s") | ("p", "r") | ("s", "p"):
        print("Win")
    case ("r", "p") | ("p", "s") | ("s", "r"):
        print("Lose")
    case _:
        print("Bad throw")
```

☞ **Your turn:** Build a simple grade calculator. Write code that:

- Takes a numeric score (0-100)
- Returns "A" for 90+, "B" for 80-89, "C" for 70-79, "D" for 60-69, "F" for below 60
- Handles invalid inputs (negative numbers, or over 100)

Try it with different approaches - if/elif chains, then try the match statement. Which feels more readable to you?

7.2. Loops

Use the `range()` function to create a `for` loop that executes a specific number of times:

```
for x in range(3):
    print(x)
>>> 0
>>> 1
>>> 2
```

Pass two arguments to `range()` to specify start and stop values (the loop will go up to, but not include, the stop value):

```
for x in range(5, 8):
    print(x)
>>> 5
>>> 6
>>> 7
```

Pass three arguments to `range()` to specify start, stop, and step values:

```
for x in range(1, 10, 3):
    print(x)
>>> 1
>>> 4
>>> 7
```

To count backward, use `range()` with a negative step value:

```
for x in range(5, -1, -1):
    print(x)
>>> 5
>>> 4
>>> 3
>>> 2
>>> 1
>>> 0
```

7.2.1. Looping Over Common Types

Many Python objects are iterable, meaning you can loop over their elements. This includes strings, lists, dictionaries, and sets.

```
s = "abc"
for ch in s:
    print(ch)
>>> a
>>> b
>>> c
```

To iterate over the items in a list:

```
v = ["a", "b", "c"]
for el in v:
    print(el)
>>> a
```

```
>>> b
>>> c
```

Use the `.items()` method to iterate over a dictionary, accessing both keys and values:

```
d = { 'a': 'apple', 'b': 'banana', 'c': 'cherry' }
for k,v in d.items():
    print(k, v)
>>> a apple
>>> b banana
>>> c cherry
```

7.2.2. Enumerate

When you need both the index and the item while iterating, use the `enumerate()` function:

```
a = ["a", "b", "c"]
for idx, item in enumerate(a):
    print(f"{idx} -> {item}")
>>> 0 -> a
>>> 1 -> b
>>> 2 -> c
```

Chapter 8. Functions

Functions are one of the core building blocks of any language. Here are some examples of how to use functions in Python

8.1. Basic Function Definition

The simplest function looks like this:

```
def greet():
    print("Hello, World")

greet()  # Call the function
```

Functions are defined with the `def` keyword, followed by the function name and parentheses. The colon starts the function body, the content of the function must be indented.

8.2. Functions with Parameters

To pass a parameter as input to a function:

```
def greet_person(name):
    print(f"Hello, {name}")

greet_person("Alice")
>>> Hello, Alice
```

A function with multiple parameters:

```
def intro(name, city):
    print(f"Hi, I'm {name} from {city}")

intro("Sarah", "New York")
>>> Hi, I'm Sarah from New York
```

Default parameters are specified with the parameter name:

```
def announce_score(team="Home", score=0):
    print(f"{team} team has scored {score} points")

# Usage examples
announce_score("Warriors", 102)
>>> Warriors team has scored 102 points

announce_score("Lakers")
>>> Lakers team has scored 0 points
```

```
announce_score()
>>> Home team has scored 0 points
```

A parameter with a default value is an optional parameter, a parameter without a default value is a required parameter. You can mix required and optional parameters, but optional parameters must come last:

```
def create_user(username, email, role="user", active=True):
    return {
        "username": username,
        "email": email,
        "role": role,
        "active": active
    }

user1 = create_user("alice", "alice@example.com")
```

If you call a function without providing all required arguments, Python will raise a `TypeError`:

```
def intro(name, city):
    print(f"Hi, I'm {name} from {city}")

intro("Sarah")

>>> Traceback (most recent call last):
>>>  File "<stdin>", line 1, in <module>
>>> TypeError: intro() missing 1 required positional argument: 'city'
```

☞ **Your Turn**: Before we move on, try writing a function called `describe_pet()` that takes a pet's name and animal type, with animal type defaulting to "dog". Make it return a string like "Buddy is a friendly dog."

8.3. Return Values

Use the `return` statement to return values from functions:

```
def add(a, b):
    return a + b

result = add(5, 3)
print(result)
>>> 8
```

Without an explicit return, functions return None:

```
def say_hello():
    print("Hello")

result = say_hello()
>>> Hello

print(result)
>>> None
```

8.4. Multiple Return Values

Python makes it easy to return multiple values using tuples:

```
def get_name_parts(full_name):
    parts = full_name.split()
    first_name = parts[0]
    last_name = parts[-1]
    return first_name, last_name

first, last = get_name_parts("John Doe")
print(f"First: {first}, Last: {last}")
>>> First: John, Last: Doe
```

8.5. Keyword Arguments

You can call functions using parameter names, this makes code easier to read since you don't have to remember what order the arguments are in:

```
def create_rectangle(width, height, color="blue"):
    return f"A {color} rectangle: {width}x{height}"

# These are all equivalent:
rect1 = create_rectangle(10, 5)
rect2 = create_rectangle(width=10, height=5)
rect3 = create_rectangle(height=5, width=10)  # Order doesn't matter with
keywords
rect4 = create_rectangle(10, height=5, color="red")  # Mix positional and
keyword
```

8.6. Variable Arguments (*args)

Sometimes you don't know how many arguments you'll get, use * before the parameter name to accept any number of positional arguments, this creates a tuple of arguments assigned to the

parameter name.

```
def sum_all(*numbers):
    total = 0
    for num in numbers:
        total += num
    return total

print(sum_all(1, 2, 3))        # Output: 6
print(sum_all(1, 2, 3, 4, 5))  # Output: 15
print(sum_all())               # Output: 0
```

8.7. Keyword Variable Arguments (**kwargs)

For flexible keyword arguments use two ** with the parameter name, this creates a dictionary of keyword arguments assigned to the parameter name.

```
def create_profile(**info):
    profile = {}
    for key, value in info.items():
        profile[key] = value
    return profile

user = create_profile(name="Alice", age=30, city="Boston", job="Engineer")
print(user)
>>> {'name': 'Alice', 'age': 30, 'city': 'Boston', 'job': 'Engineer'}
```

You can combine all argument types, but I wouldn't recommend it. To make code easiest to understand and most flexible, use keyword arguments.

```
def flexible_function(required_arg, default_arg="default", *args, **
kwargs):
    print(f"Required: {required_arg}")
    print(f"Default: {default_arg}")
    print(f"Extra args: {args}")
    print(f"Keyword args: {kwargs}")

flexible_function("hello", "world", 1, 2, 3, name="Alice", age=25)

>>> Required: hello
>>> Default: world
>>> Extra args: (1, 2, 3)
>>> Keyword args: {'name': 'Alice', 'age': 25}
```

8.8. Lambda Functions

For simple, one-line functions, you can use a lambda function:

```
# Regular function
def square(x):
    return x * x

# Lambda equivalent
square_lambda = lambda x: x * x

print(square(5))        # Output: 25
print(square_lambda(5)) # Output: 25
```

Lambdas are most useful with functions like `map()`, `filter()`, and `sorted()`:

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

# Square all numbers
squared = list(map(lambda x: x * x, numbers))
print(squared)
>>> [1, 4, 9, 16, 25]

# Filter even numbers
evens = list(filter(lambda x: x % 2 == 0, numbers))
print(evens)
>>> [2, 4]

# Sort by custom criteria
people = [("Alice", 25), ("Bob", 30), ("Charlie", 20)]
sorted_by_age = sorted(people, key=lambda person: person[1])
print(sorted_by_age)
>>> [('Charlie', 20), ('Alice', 25), ('Bob', 30)]
```

8.9. Function Documentation

Document functions using docstrings:

```
def calculate_compound_interest(principal, rate, time, compound_frequency=
1):
    """
    Calculate compound interest.

    Args:
        principal (float): Initial amount of money
        rate (float): Annual interest rate (as decimal, e.g., 0.05 for 5%)
```

```
        time (float): Time in years
        compound_frequency (int): How many times interest compounds per
year

    Returns:
        tuple: (final_amount, interest_earned)

    Example:
        >>> amount, interest = calculate_compound_interest(1000, 0.05, 2)
        >>> print(f"Amount: ${amount:.2f}, Interest: ${interest:.2f}")
    """
    final_amount = principal * (1 + rate/compound_frequency) **
(compound_frequency * time)
    interest_earned = final_amount - principal
    return final_amount, interest_earned

# Usage
amount, interest = calculate_compound_interest(1000, 0.05, 2, 4)
print(f"Final amount: ${amount:.2f}")
print(f"Interest earned: ${interest:.2f}")
```

8.10. Nested Functions

You can define functions inside other functions. A primary reason for this is to create closures. A **closure** allows the inner function to remember and access variables from the outer function's scope, even after the outer function has finished executing. This pattern is useful for creating function factories or specialized functions tailored by parameters passed to the outer function.

Here's a practical example of a closure - creating a counter function:

```
def create_counter(start=0):
    count = start

    def counter():
        nonlocal count
        count += 1
        return count

    return counter

# Create different counters
page_counter = create_counter(1)
error_counter = create_counter(0)

print(page_counter())  # Output: 2
print(page_counter())  # Output: 3
print(error_counter()) # Output: 1
print(page_counter())  # Output: 4
```

The inner `counter` function "closes over" the `count` variable from the outer function's scope. Each time we call `create_counter()`, it creates a new closure with its own `count` variable that persists between calls to the returned function.

8.11. Tips on Using Functions

Here are the patterns I aim to follow when writing functions:

1. **Keep functions small and focused** - each function should do one thing well
2. **Use descriptive names** - `calculate_tax()` is better than `calc()`
3. **Return early when possible** - avoid deep nesting
4. **Use type hints** for clarity, see Type Hints chapter

Chapter 9. Classes

Classes are Python's way of creating your own custom data types, used to bundle related data and functions together. Python classes provide all the standard features of Object Oriented Programming.

9.1. Basic Class Definition

Here's a simple class example:

```
class Person:
    def __init__(self, name):
        self.name = name

# Create an instance
person = Person("Marcus")
person.name
>>> Marcus
```

The *init* function is the constructor for the class. It is called automatically when you create a new instance of the class. The `self` parameter refers to the instance itself and is used to initialize the instance variables.

9.2. Class wtih Default Values

Here's an example of a class with default values:

```
class Person:
    def __init__(self, name="Unknown"):
        self.name = name
```

So when creating instances, if you do not provide a name it will set the default value.

```
alice = Person("Alice")
billie = Person()

print(alice.name)   # Output: Alice
print(billie.name)  # Output: Unknown
```

9.3. Class vs Instance Variables

There are two types of variables in classes: class variables (shared by all instances) and instance variables (unique to each instance).

```
class Dog:
```

```
    # Class variable - shared by all dogs
    species = "Canis lupus"

    def __init__(self, name, breed):
        # Instance variables - unique to each dog
        self.name = name
        self.breed = breed

buddy = Dog("Buddy", "Golden Retriever")
max_dog = Dog("Max", "German Shepherd")

print(buddy.species)    # Output: Canis lupus
print(max_dog.species)  # Output: Canis lupus
print(buddy.name)       # Output: Buddy
print(max_dog.name)     # Output: Max

# Changing class variable affects all instances
Dog.species = "Domestic Dog"
print(buddy.species)    # Output: Domestic Dog
print(max_dog.species)  # Output: Domestic Dog
```

9.4. Methods

Methods are functions that belong to a class. The first parameter is always `self`, which refers to the instance:

```
class BankAccount:
    def __init__(self, owner, balance=0):
        self.owner = owner
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount
        return self.balance

    def withdraw(self, amount):
        if amount <= self.balance:
            self.balance -= amount
            return self.balance
        else:
            return "Insufficient funds"

    def get_balance(self):
        return self.balance

# Using the class
account = BankAccount("Alice", 100)
print(account.get_balance())  # Output: 100
```

```
account.deposit(50)
print(account.get_balance())  # Output: 150

account.withdraw(30)
print(account.get_balance())  # Output: 120

print(account.withdraw(200))  # Output: Insufficient funds
```

9.5. String Representation

If you try to print an instance of a class, by default it won't output anything meaningful. For example,

```
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author

p = Person("Cujo", "Stephen King")
print(p)
>>> <__main__.Book object at 0x10063b200>
```

To show a human-readable representation, you need to define the `__str__` method.

```
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author

    def __str__(self):
        return f"{self.title} by {self.author}"

p = Book("Cujo", "Stephen King")
print(p)
>>> Cujo by Stephen King
```

There is also `__repr__` for developers and debugging, typically `__repr__` is used to provide a string representation of an object that can be used to recreate the object. If `__str__` is not defined, `__repr__` is used as a fallback.

```
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author
```

```
    def __str__(self):
        return f"{self.title} by {self.author}"

    def __repr__(self):
        return f"Book('{self.title}', '{self.author}')"

b = Book("Cujo", "Stephen King")
print(b)        # Output: Cujo by Stephen King
print(str(b))   # Output: Cujo by Stephen King
print(repr(b))  # Output: Book('Cujo', 'Stephen King')

r = repr(b)
nb = eval(r)    # Create new book object from output
print(nb)       # Output: Cujo by Stephen King
```

Summary: `__str__` is for human-readable output, `__repr__` is for developers and debugging.

9.6. Property Decorators

Properties let you access methods like attributes. use them for computed values or to add validation:

```
class Circle:
    def __init__(self, radius):
        self._radius = radius

    @property
    def radius(self):
        return self._radius

    @radius.setter
    def radius(self, value):
        if value < 0:
            raise ValueError("Radius cannot be negative")
        self._radius = value

    @property
    def area(self):
        return 3.14159 * self._radius ** 2

    @property
    def circumference(self):
        return 2 * 3.14159 * self._radius

circle = Circle(5)
print(circle.radius)        # Output: 5
print(circle.area)          # Output: 78.53975
print(circle.circumference) # Output: 31.4159
```

```
circle.radius = 10
print(circle.area)          # Output: 314.159

# circle.radius = -5  # Would raise ValueError
```

9.7. Class Methods and Static Methods

At times you may want to create a class to create methods that work with the class itself, not instances:

```
class MathUtils:
    pi = 3.14159

    @classmethod
    def circle_area(cls, radius):
        return cls.pi * radius ** 2

    @staticmethod
    def add(a, b):
        return a + b

    @staticmethod
    def is_even(number):
        return number % 2 == 0

# Class methods can be called on the class
area = MathUtils.circle_area(5)
print(area)  # Output: 78.53975

# Static methods are just regular functions grouped with the class
result = MathUtils.add(10, 20)
print(result)  # Output: 30

print(MathUtils.is_even(4))  # Output: True
print(MathUtils.is_even(7))  # Output: False
```

9.8. Inheritance

Inheritance lets you create new classes based on existing ones:

```
class Animal:
    def __init__(self, name, species):
        self.name = name
        self.species = species

    def speak(self):
        return f"{self.name} makes a sound"
```

```
    def info(self):
        return f"{self.name} is a {self.species}"

class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__(name, "Dog")  # Call parent constructor
        self.breed = breed

    def speak(self):  # Override parent method
        return f"{self.name} barks!"

    def fetch(self):  # New method specific to dogs
        return f"{self.name} fetches the ball"

class Cat(Animal):
    def __init__(self, name, color):
        super().__init__(name, "Cat")
        self.color = color

    def speak(self):
        return f"{self.name} meows!"

    def climb(self):
        return f"{self.name} climbs the tree"

# Using inheritance
buddy = Dog("Buddy", "Golden Retriever")
whiskers = Cat("Whiskers", "Orange")

print(buddy.info())    # Output: Buddy is a Dog
print(buddy.speak())   # Output: Buddy barks!
print(buddy.fetch())   # Output: Buddy fetches the ball

print(whiskers.info()) # Output: Whiskers is a Cat
print(whiskers.speak()) # Output: Whiskers meows!
print(whiskers.climb()) # Output: Whiskers climbs the tree
```

9.9. Dunder Methods

The `__str__` and `__repr__` earlier are dunder methods (double underscore) that are special methods that are automatically called by Python in various contexts. There are several other dunder methods that you can define in your classes to customize their behavior.

9.9.1. Operators

The following operator examples will use the `Point` class to demonstrate.

```
class Point:
```

```
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

Equality: Define `__eq__` method to compare two objects using the `==` operator:

```
    def __eq__(self, other):
        if isinstance(other, Point):
            # return True only if both x and y are equal
            return self.x == other.x and self.y == other.y
        return False

p1 = Point(2,4)
p2 = Point(3,5)
p3 = Point(2,4)
print(p1 == p2)     # Output: False
print(p1 == p3)     # Output: True
```

Math Operators: Define `__add__` method on a class to define what it means to add two objects together using the `+` operator:

```
    def __add__(self, other):
        if isinstance(other, Point):
            return Point(self.x + other.x, self.y + other.y)
        raise TypeError("Can only add Point objects")

p1 = Point(2,4)
p2 = Point(3,5)
print(p1 + p2)      # Output: Point(5, 9)
```

You can also add methods for the following math operators and symbols:

Method	Symbol	Description
`__sub__`	`-`	Subtraction
`__mul__`	`*`	Multiplication
`__truediv__`	`/`	Division
`__floordiv__`	`//`	Floor Division
`__mod__`	`%`	Modulus
`__pow__`	`**`	Exponentiation

Logic Operators: Additional logic operators are defined in the same way:

Method	Symbol	Description

`__lt__`	`<`	Less than
`__le__`	`⇐`	Less than or equal to
`__gt__`	`>`	Greater than
`__ge__`	`>=`	Greater than or equal to
`__ne__`	`!=`	Not equal to
`__and__`	`&`	Logical AND
`__or__`	`|`	Logical OR
`__invert__`	`~`	Logical NOT

I don't recommend overloading operators too much, it can be fun to do so, but it can also lead to confusion and errors if not used carefully.

9.9.2. Iterator

You can build your own iterator to use in a loop. To do so you must implement the *`iter`* and *`next`* methods.

```
class Doubler:
    def __init__(self, start, maxim=100):
        self.num= start / 2
        self.maxim = maxim

    def __iter__(self):
        return self

    def __next__(self):
        self.num = self.num * 2
        if self.num > self.maxim:
            raise StopIteration
        return int(self.num)

print("Doubling from 2 to 100")
for i in Doubler(2):
    print(i)

print("Doubling from 3 to 200")
for i in Doubler(3, 200):
    print(i)
```

9.9.3. Generator

For simple iterators it is often easier to use a generator function to achieve the same result. A generator is a regular functions that use the `yield` keyword to return a value. Each time `next()` is called on it, the generator will resume execution from where it left off. What makes generators easier to use is the `__iter__()` and `__next__()` methods are created automatically.

```
def doubler(num, maxim=100):
    while num <= maxim:
        yield int(num)
        num *= 2

print("Doubling from 2 to 100")
for i in doubler(2):
    print(i)

print("Doubling from 3 to 200")
for i in doubler(3, 200):
    print(i)
```

9.10. Class Summary

Classes are incredibly powerful for organizing your code and modeling real-world concepts. Start with simple classes and gradually add complexity as you need it. The key is to think about what data and behaviors naturally belong together.

Chapter 10. Python Idioms

Python has many idiomatic patterns that make code more readable, maintainable, and "*Pythonic*". Here are some of the essential idioms every new Python developer should know.

10.1. The `if __name__ == "__main__"` Pattern

This is one of the most important Python idioms you'll encounter:

```
def main():
    print("This script is being run directly")
    # Your main program logic here

if __name__ == "__main__":
    main()
```

10.1.1. Why Use This Pattern?

When Python executes a script, it sets a special built-in variable called `__name__` before running any code. The value of `__name__` depends on how the script is being used:

- If you run your file directly (e.g., by typing `python your_script.py` in the terminal), Python sets `__name__` to the string `"__main__"`.
- If your file is imported as a module into another script (e.g., `import your_script`), Python sets `__name__` to the name of the module (e.g., `"your_script"` if your file is named `your_script.py`).

This pattern allows your script to:

1. **Be both a script and a module** - Code can be imported without executing the main logic
2. **Prevent unwanted execution** - When someone imports your module, the main code won't run
3. **Enable testing** - You can import functions for testing without side effects

10.1.2. Example:

```
# calculator.py
def add(a, b):
    return a + b

def subtract(a, b):
    return a - b

def main():
    print("Calculator running...")
    result = add(5, 3)
    print(f"5 + 3 = {result}")
```

```
if __name__ == "__main__":
    main()
```

Now you can either run it directly (`python calculator.py`) or import it (`from calculator import add`).

10.2. Essential Python Idioms

10.2.1. List Comprehensions

Replace verbose loops with concise list comprehensions:

```
# Instead of this:
squares = []
for i in range(10):
    squares.append(i ** 2)

# Use this:
squares = [i ** 2 for i in range(10)]

# With conditions:
even_squares = [i ** 2 for i in range(10) if i % 2 == 0]
```

10.2.2. Dictionary Comprehensions

Create dictionaries efficiently:

```
# Create a dictionary from two lists
keys = ['a', 'b', 'c']
values = [1, 2, 3]
my_dict = {k: v for k, v in zip(keys, values)}

# Transform existing dictionary
prices = {'apple': 0.50, 'banana': 0.30, 'orange': 0.75}
expensive = {fruit: price for fruit, price in prices.items() if price >
0.40}
```

10.2.3. Enumerate Instead of Range(len())

```
items = ['apple', 'banana', 'cherry']

# Instead of this:
for i in range(len(items)):
    print(f"{i}: {items[i]}")
```

```
# Use this:
for i, item in enumerate(items):
    print(f"{i}: {item}")
```

10.2.4. Zip for Parallel Iteration

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

# Instead of indexing:
for i in range(len(names)):
    print(f"{names[i]} is {ages[i]} years old")

# Use zip:
for name, age in zip(names, ages):
    print(f"{name} is {age} years old")
```

10.2.5. Context Managers (with statements)

Always use context managers for resource management:

```
# File handling
with open('file.txt', 'r') as f:
    content = f.read()
# File automatically closed

# Custom context managers
from contextlib import contextmanager

@contextmanager
def timer():
    import time
    start = time.time()
    yield
    end = time.time()
    print(f"Elapsed: {end - start:.2f} seconds")

with timer():
    # Your code here
    time.sleep(1)
```

10.2.6. String Formatting

Use f-strings (Python 3.6+) for readable string formatting, see String Formatting for more:

```
name = "Alice"
age = 30

# Old ways (avoid):
message = "Hello, %s! You are %d years old." % (name, age)
message = "Hello, {}! You are {} years old.".format(name, age)

# Modern way:
message = f"Hello, {name}! You are {age} years old."

# With expressions:
price = 19.99
message = f"Total: ${price * 1.08:.2f}"  # Includes tax, 2 decimal places
```

10.2.7. Default Dictionary Values

Use `dict.get()` or `collections.defaultdict`:

```
my_dict = {'item1': 'apple', 'item2': 'banana'}

# Instead of checking if key exists:
if 'key' in my_dict:
    value = my_dict['key']
else:
    value = 'default'

# Use get():
value = my_dict.get('key', 'default') # Returns 'default' if 'key' is not
in my_dict
value_present = my_dict.get('item1', 'default') # Returns 'apple'

# For counting or grouping:
from collections import defaultdict

# Count items in a list
fruits = ['apple', 'banana', 'apple', 'orange', 'banana', 'apple']
fruit_counter = defaultdict(int)
for fruit in fruits:
    fruit_counter[fruit] += 1
# fruit_counter will be defaultdict(<class 'int'>, {'apple': 3, 'banana':
2, 'orange': 1})

# Group items by a category
class Item:
    def __init__(self, name, category):
        self.name = name
        self.category = category
```

```
items_to_group = [
    Item('apple', 'fruit'),
    Item('banana', 'fruit'),
    Item('carrot', 'vegetable'),
    Item('broccoli', 'vegetable'),
    Item('orange', 'fruit')
]

grouped_items = defaultdict(list)
for item in items_to_group:
    grouped_items[item.category].append(item.name)
# grouped_items will be defaultdict(<class 'list'>, {'fruit': ['apple',
'banana', 'orange'], 'vegetable': ['carrot', 'broccoli']})
```

10.2.8. Unpacking and Multiple Assignment

```
# Tuple unpacking
point = (3, 4)
x, y = point

# Swap variables
a, b = b, a

# Extended unpacking (Python 3+)
first, *middle, last = [1, 2, 3, 4, 5]
# first = 1, middle = [2, 3, 4], last = 5

# Function arguments
def greet(name, age, city):
    return f"Hi {name}, {age} years old from {city}"

person_data = ("Alice", 30, "New York")
message = greet(*person_data)  # Unpacking tuple

person_dict = {"name": "Bob", "age": 25, "city": "Boston"}
message = greet(**person_dict)  # Unpacking dictionary
```

10.2.9. Exception Handling Best Practices

```
# Be specific with exceptions
try:
    value = int(user_input)
except ValueError:
    print("Please enter a valid number")
except KeyboardInterrupt:
```

```
    print("Operation cancelled")

# Use else and finally
try:
    file = open('data.txt')
except FileNotFoundError:
    print("File not found")
else:
    # Runs only if no exception occurred
    data = file.read()
finally:
    # Always runs
    if 'file' in locals():
        file.close()

# EAFP (Easier to Ask for Forgiveness than Permission)
try:
    return my_dict[key]
except KeyError:
    return default_value
```

10.2.10. Generator Expressions and Functions

Use generators for memory-efficient iteration:

```
# Generator expression (like list comprehension but lazy)
squares_gen = (i ** 2 for i in range(1000000))  # Memory efficient

# Generator function
def fibonacci():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

# Use generators
fib = fibonacci()
first_10 = [next(fib) for _ in range(10)]
```

10.2.11. Pathlib for File Operations

Use `pathlib` instead of `os.path`, see Files for more:

```
from pathlib import Path

# Instead of os.path
```

```
import os
file_path = os.path.join('data', 'files', 'input.txt')
if os.path.exists(file_path):
    with open(file_path, 'r') as f:
        content = f.read()

# Use pathlib
file_path = Path('data') / 'files' / 'input.txt'
if file_path.exists():
    content = file_path.read_text()

# More pathlib examples
current_dir = Path.cwd()
parent_dir = Path(__file__).parent
all_py_files = list(Path('.').glob('**/*.py'))
```

10.2.12. Type Hints (Python 3.5+)

Add type hints for better code documentation and IDE support, see Type Hints for more:

```
from typing import List, Dict, Optional, Union

def process_items(items: List[str], multiplier: int = 1) -> List[str]:
    """Process a list of items."""
    return [item * multiplier for item in items]

def get_user_data(user_id: int) -> Optional[Dict[str, Union[str, int]]]:
    """Get user data, returns None if not found."""
    # Implementation here
    pass
```

10.2.13. Truthy and Falsy Values

Understand how Python evaluates values in a boolean context:

```
# Falsy values:
# - None
# - False
# - Zero of any numeric type (0, 0.0, 0j)
# - Empty sequences (list, tuple, string: [], (), '')
# - Empty mappings (dictionary: {})
# - Instances of user-defined classes, if the class defines a __bool__() or
__len__() method that returns False or 0.

my_list = []
if not my_list:  # More Pythonic than if len(my_list) == 0:
    print("List is empty")
```

```
my_string = "Hello"
if my_string:  # True because it's not empty
    print(f"String: {my_string}")
```

10.2.14. Iterable Slicing

Work with parts of sequences efficiently:

```
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

first_three = numbers[:3]  # [0, 1, 2]
last_three = numbers[-3:] # [7, 8, 9]
middle = numbers[2:7]   # [2, 3, 4, 5, 6]
every_other = numbers[::2] # [0, 2, 4, 6, 8]
reverse_list = numbers[::-1] # [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]

# Slicing works on strings too
text = "Pythonic"
print(text[1:4]) # yth
```

10.2.15. Use Underscore as a Throwaway Variable

Use _ for variables whose values you don't intend to use:

```
# When unpacking, if you only need certain values
coordinates = (10, 20, 5)
x, _, z = coordinates # y is ignored

# In loops where the index or value is not needed
for _ in range(5):
    print("Hello")
```

10.2.16. Use `any()` and `all()`

Check for conditions in iterables concisely:

```
numbers = [1, 2, 3, -5, 6]

# Check if any number is negative
has_negative = any(n < 0 for n in numbers)
print(f"Has negative numbers: {has_negative}") # True

# Check if all numbers are positive
all_positive = all(n > 0 for n in numbers)
```

```
print(f"All numbers positive: {all_positive}") # False
```

10.3. Code Style and Standards

10.3.1. Follow PEP 8

Use tools to enforce Python's style guide, I recommend using `ruff` for linting and formatting. An extension is available for popular editors like VSCode. See Dev Environment for more.

To install with `pip`and use `ruff`, follow these steps:

```
# Install ruff
pip install ruff

# Check style
ruff check

# Format code
ruff format
```

10.3.2. Naming Conventions

```
# Variables and functions: snake_case
user_name = "Alice"
def calculate_total_price():
    pass

# Constants: UPPER_SNAKE_CASE
MAX_CONNECTIONS = 100
API_BASE_URL = "https://api.example.com"

# Classes: PascalCase
class UserAccount:
    pass

# Private attributes: leading underscore
class MyClass:
    def __init__(self):
        self._private_var = "internal use"  # Convention: for internal use,
still accessible
        self.__very_private = "name mangled" # Name mangling: becomes
_MyClass__very_private
```

10.3.3. Documentation

Write clear docstrings:

```
def calculate_area(length: float, width: float) -> float:
    """Calculate the area of a rectangle.

    Args:
        length: The length of the rectangle
        width: The width of the rectangle

    Returns:
        The area of the rectangle

    Raises:
        ValueError: If length or width is negative
    """
    if length < 0 or width < 0:
        raise ValueError("Length and width must be non-negative")
    return length * width
```

10.4. Testing Your Code

Write tests using pytest, see <<Testing> for more.

```
# test_calculator.py
import pytest
from calculator import add, subtract

def test_add():
    assert add(2, 3) == 5
    assert add(-1, 1) == 0

def test_subtract():
    assert subtract(5, 3) == 2
    assert subtract(0, 5) == -5

def test_add_with_floats():
    assert add(0.1, 0.2) == pytest.approx(0.3)

# Run tests
# pytest test_calculator.py
```

Chapter 11. Type Hints

In Python 3.5, **type hints** were introduced as a means for adding type information. Starting with Python 3.9, many of these types are available as built-ins, so you no longer need to import them from the `typing` module. This page assumes you are using Python 3.9+.

Think of type hints primarily as a documentation tool or a helper for IDEs and static analysis. The Python interpreter does not enforce type hints at runtime, but they provide valuable information for you as a developer and for various tools.

The primary benefit of using type annotations is to document your code and catch potential bugs early. They are particularly helpful with complex data structures, for example, when making a list of tuples.

```
my_list_of_tuples: list[tuple[str, int]] = []
```

11.1. Basic Types

The core basic types: `str`, `int`, `float`, and `bool`.

```
my_string: str = ""
my_int: int = 0
my_float: float = 0.0
my_bool: bool = True
```

The core collection types: `list`, `dict`, `tuple`, and `set`.

```
my_list: list[str] = []
my_dict: dict[str, int] = {}
my_tuple: tuple[str, int] = ("hello", 42)  # Fixed-length tuple
my_set: set[int] = set()
```

For variable-length tuples, use ellipsis:

```
my_var_tuple: tuple[int, ...] = (1, 2, 3, 4, 5)  # Variable-length tuple
```

11.2. Functions

To use type hints in a function for arguments and return values.

```
def add(a: int, b: int) -> int:
    return a + b
```

Use `None` when your function does not return a value.

```
def hello(name: str) -> None:
    print(f"Hello {name}")
```

Functions with default arguments:

```
def greet(name: str, greeting: str = "Hello") -> str:
    return f"{greeting}, {name}!"
```

11.3. Classes

To use type hints in a class.

```
class Point:
    x: int
    y: int

    def __init__(self, x: int, y: int) -> None:
        self.x = x
        self.y = y

    def distance_from_origin(self) -> float:
        return (self.x ** 2 + self.y ** 2) ** 0.5
```

11.4. Custom Types

You can use your own classes as type hints in other places:

```
def move_point(point: Point, dx: int, dy: int) -> Point:
    return Point(point.x + dx, point.y + dy)

my_point: Point = Point(1, 2)
new_point: Point = move_point(my_point, 3, 4)
```

11.5. Optional Types

For variables that can be `None`, use the union syntax (`|`) with `None`.

```
# Modern syntax (Python 3.10+)
my_optional_string: str | None = None
```

For older Python versions, you can still import `Optional`:

```
from typing import Optional

my_optional_string: Optional[str] = None
```

11.6. Union Types

Use the `|` operator for variables that can hold values of multiple types (Python 3.10+):

```
# Modern syntax
my_union: int | str = "Hello"
my_union = 123

# Multiple types
id_value: int | str | None = None
```

For older Python versions:

```
from typing import Union

my_union: Union[int, str] = "Hello"
```

11.7. Literal Types

Use `Literal` for values that must be specific literals:

```
from typing import Literal

def set_mode(mode: Literal["read", "write", "append"]) -> None:
    print(f"Setting mode to {mode}")

# This will work
set_mode("read")

# This will cause a type checker error
set_mode("invalid")
```

11.8. Any Type

Use `Any` when you need to opt out of type checking. Try to limit its use, as it bypasses type checking.

```python
from typing import Any

def process_data(data: Any) -> Any:
    # This function can accept and return anything
    return data
```

11.9. Type Aliases

Create type aliases for complex types:

```python
# Modern syntax using built-in types
Coordinates = list[tuple[float, float]]
UserID = int
JSONData = dict[str, Any]

my_coordinates: Coordinates = [(1.0, 2.0), (3.0, 4.0)]
user_id: UserID = 12345
```

11.10. Generics and TypeVar

Use generics and `TypeVar` to create generic functions and classes:

```python
from typing import TypeVar, Generic

T = TypeVar('T')

def first_item(items: list[T]) -> T | None:
    return items[0] if items else None

# Usage
numbers = [1, 2, 3]
first_num = first_item(numbers)  # Type checker knows this is int | None

strings = ["a", "b", "c"]
first_str = first_item(strings)  # Type checker knows this is str | None
```

Generic classes:

```python
class Stack(Generic[T]):
    def __init__(self) -> None:
        self._items: list[T] = []

    def push(self, item: T) -> None:
        self._items.append(item)
```

```
    def pop(self) -> T | None:
        return self._items.pop() if self._items else None

# Usage
int_stack: Stack[int] = Stack()
str_stack: Stack[str] = Stack()
```

11.11. Final Types

Use `Final` for constants that should not be reassigned:

```
from typing import Final

MAX_CONNECTIONS: Final = 100
API_VERSION: Final[str] = "v1.2.3"

# This would cause a type checker error:
MAX_CONNECTIONS = 200
```

11.12. When to Use Type Hints

Type hints are most beneficial when you are: - Working with complex data structures - Building libraries or APIs that others will use - Working in teams where code clarity is important - Wanting better IDE support and error detection

You might skip type hints for: - Simple scripts or one-off utilities - Prototyping or exploratory code - Situations where the types are obvious from context

11.13. Tools

Most editors these days have support for type hints to provide better autocomplete and error checking. Popular type checkers include:

11.13.1. mypy

The original and most widely used type checker:

```
# Install mypy
pip install mypy

# Run mypy on a file
mypy my_file.py

# Run mypy on a directory
mypy src/
```

11.13.2. Pyright/Pylance

Microsoft's type checker, used in VS Code's Pylance extension:

```
# Install pyright
npm install -g pyright

# Run pyright
pyright src/
```

11.13.3. Configuration

You can configure type checkers with configuration files:

mypy.ini or **pyproject.toml** for mypy:

```
[mypy]
python_version = 3.9
warn_return_any = True
warn_unused_configs = True
disallow_untyped_defs = True
```

pyrightconfig.json for Pyright:

```
{
    "pythonVersion": "3.9",
    "typeCheckingMode": "strict",
    "reportMissingImports": true
}
```

Chapter 12. Collections

Aside from Lists and Dictionaries, Python has several other collection objects that are useful in the standard `collections` module.

See: Python Collections documentation

12.1. Sets

Python `set` object is equivalent to a mathematical set, and supports similar operations. Think of a `set` as a `list`, but every element is unique and a set is unordered.

Note: I know sets are not in the collections module but a built-in, it just felt like this section of the guide made the most sense.

12.1.1. Create a set

Create a set multiple ways, when creating a set it will automatically drop any duplicate elements.

```
# using curly brackets
A = { 'a', 'b', 'c', 'a'}
>>> { 'c', 'a', 'b' }

# from list
A = set([1, 2, 3, 1])
>>> {1, 2, 3}

# using comprehensions
A = { c for c in 'abracadabra' }
>>> {'d', 'c', 'b', 'a', 'r'}
```

12.1.2. Set Operations

Perform common set operations, union, intersection and difference.

```
A = { 1, 3, 5, 7, 9 }
B = { 2, 4, 6, 8, 0 }
C = { 2, 3, 4 }

# return set that combines the two sets
A.union(B)
>>> {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

# return set with elements common between two sets
A.intersection(C)
>> {3}

# return set with elements only in A, not C
```

```
A.difference(C)
>>> {1, 5, 9, 7}
```

The operations also work on an arbritary number of sets.

```
A = set() # empty set
B = { 2, 3 }
C = { 4, 5 }
D = { 6, 7 }
A.union(B, C, D)
>>> { 2, 3, 4, 5, 6, 7 }
```

12.2. Default Dictionary

The `defaultdict` is extremely useful when building a dictionary and you don't want to always check if it already contains a key.

Here's how you might count characters in a string using the standard `dict` object.

```
d = {}
s = "abcdadkacb"
for ch in s:
    if not ch in d:
        d[ch] = 1
    else:
        d[ch] += 1
```

Here's an easier way using `defaultdict` type and avoiding the extra check.

```
from collections import defaultdict

d = defaultdict(int)
s = "abcdadkacb"
for ch in s:
    d[ch] += 1
```

Using `int` as the default will always return 0 for missing keys. Set your own default using a lambda function returning a constant. For example to have a default value of 1 use:

```
from collections import defaultdict
d = defaultdict(lambda: 1)
keys = 'abc'
d['b'] = 2
for k in keys:
```

```
    print(d[k])
>>> 1
>>> 2
>>> 1
```

12.3. Named Tuple

The `namedtuple` is a convenience type that allows you to create a tuple and assign names for the positions that can then be accessed. This is useful to make it more readable and simpler to access.

```
from collections import namedtuple

Point = namedtuple('Point', ['x', 'y'])
p1 = Point(1, 2)
p2 = Point(y=4, x=2)

d = abs(p2.x - p1.x) + abs(p2.y - p1.y)
```

12.4. deque

The `deque` object (pronounced "deck") is an efficient double-sided list. It adds methods `.popleft()` and `.appendleft()` to add and remove items from both left and right sides of list.

```
from collections import deque

d = deque(['a', 'b', 'c'])
a = d.popleft()
d.appendleft('z')
print(d)
>>> deque(['z', 'b', 'c'])
```

12.4.1. Rotation

A `deque` object additional has a `.rotate(n)` method which allows rotating the elements n times, this works for both positive and negative values of n.

```
from collections import deque

d = deque(['a', 'b', 'c' ,'d'])
d.rotate(2)
print(d)
>>> deque(['c', 'd', 'a', 'b'])

d.rotate(-3)
print(d)
```

```
>>> deque(['b', 'c', 'd', 'a'])
```

12.5. Counter

The `Counter` is useful to count items in a list and return a dict of value and count for each item.

```
from collections import Counter
c = Counter('abcdeabieaabi')

print(c)
>>> Counter({'a': 4, 'b': 3, 'e': 2, 'i': 2, 'c': 1, 'd': 1})
```

12.5.1. Most Common

The `.most_common(n)` method will return an ordered list of tuples of the most common items with the item and count as the two items in the tuple. c.most_common(1)

```
from collections import Counter
c = Counter('abcdeabieaabi')
mc = c.most_common(1)
print(f"Most common {mc[0][0]} with count {mc[0][1]}")
```

12.5.2. Total

Use `.total()` to return the sum of the counts.

```
from collections import Counter
c = Counter('abcdeabieaabi')
print(c.total())
```

Chapter 13. Files

13.1. Reading files

Here are several common ways to read files in Python. Use a context manager by using the `with` statement to work with files, this will automatically close the file after use.

13.1.1. Read entire file to list of lines

Use `readlines()` to read the entire file into a list, each line is an element in the list. Note that this method includes the newline characters at the end of each line.

```
with open('file.txt') as f:
    lines = f.readlines()
```

If you want a list without the newlines:

```
with open('file.txt') as f:
    lines = [x.rstrip() for x in f]
```

13.1.2. Read file in by line

The file object is iterable, so you can loop over each line to process it:

```
with open('file.txt') as f:
    for line in f:
        line = line.rstrip() # remove newline
        # process the line
```

13.1.3. Read file in to single variable

Read the entire file into a single string variable using `read()`.

```
with open('file.txt') as f:
    content = f.read()
```

13.2. Write file

To write text to a file in Python, open it in write mode (`'w'`). This creates a new file or overwrites an existing one.

```
content = "Text to write to file"
with open('file.txt', 'w') as f:
```

```
    f.write(content)
```

13.2.1. Append text to existing file

To append text to an existing file, open it in append mode (`'a'`), this writes the text to the end of the file.

```
content = "Append text to file"
with open('file.txt', 'a') as f:
    f.write(content)
```

13.3. File exists

To check if a file exists, use the `pathlib` module.

```
from pathlib import Path

p = Path("/path/to/file")
if p.is_file():
    print("Yes, the file exists")
```

13.4. Find all files matching extension

Use `pathlib.Path.glob()` to search for files matching a specific pattern. For example, to find all files with the `.md` extension in a directory:

```
from pathlib import Path

pages = Path("/path/to/dir").glob("*.md")
```

Note: Using `*.md` only searches the specified directory. To recursively search all subdirectories, use `**/*.md`:

```
from pathlib import Path

pages = Path("/path/to/dir").glob("**/*.md")
```

13.5. Copy files and directories

To copy files and directories, use the `shutil` module. Remember to import it first.

```
import shutil
```

```
# To copy a file
shutil.copy2(source, destination)

# To copy a directory tree
shutil.copytree(source, destination)
```

Use `shutil.copy2()` to copy a file. This function attempts to preserve file metadata. If you need finer control over metadata or other copy operations, consult the shutil documentation for more options.

Use `shutil.copytree()` to copy an entire directory tree.

Chapter 14. Command-line Arguments

Python is my go-to tool for command-line scripts, which often require passing command-line arguments. This guide serves as a reference for command-line argument parsing in Python.

14.1. What Library to Use

Several standard libraries can parse command-line arguments in Python. The one you should typically use is the `argparse` module. It's similar in name to `optparse`, but `optparse` is an older, deprecated module.

Another module, `getopt`, also handles command-line argument parsing but is generally more complicated and requires writing more code.

For most cases, use the `argparse` module.

14.2. Basic Example

First, you may not need much, if you only want to grab a single argument and no flags or other parameters are passed in, use the `sys.argv` list. This list contains all command-line parameters.

The first element in `sys.argv` (i.e., `sys.argv[0]`) is the script's name. A parameter passed to the script will be the second element: `sys.argv[1]`.

```
import sys
if len(sys.argv) > 1:
    print(f"~ Script: {sys.argv[0]}")
    print(f"~ Arg   : {sys.argv[1]}")
else:
    print("No arguments provided")
```

Saving this as `test.py` and running it produces:

```
$ python test.py Foo
~ Script: test.py
~ Arg   : Foo
```

14.2.1. Multiple Arguments with sys.argv

Since `sys.argv` is a list, you can access blocks of arguments or slice it just like any other Python list.

To get the last argument: `sys.argv[-1]`

To get all arguments after the script name itself: `' '.join(sys.argv[1:])` To get all arguments after the first parameter: `' '.join(sys.argv[2:])`

14.3. Flag Parameters

Use `argparse` when you want to include flags (e.g., `--help`), handle optional arguments, or manage arguments with varying lengths.

14.3.1. Help and Verbose Examples

Here's a script that accepts the `--verbose` flag:

```
import argparse

parser = argparse.ArgumentParser(description='Demo')
parser.add_argument('--verbose', action='store_true', help='verbose flag')

args = parser.parse_args()

if args.verbose:
    print("~ Verbose!")
else:
    print("~ Not so verbose")
```

Here's how to run the above example:

```
$ python test.py
~ Not so verbose

$ python test.py --verbose
~ Verbose!
```

The `action` parameter tells `argparse` to store `True` if the flag is present, otherwise it stores `False`. A great benefit of using `argparse` is the built-in help. You can see this by passing an unknown parameter, or by using `-h` or `--help`.

```
$ python test.py --help
usage: test.py [-h] [--verbose]

Demo

optional arguments:
  -h, --help  show this help message and exit
  --verbose   verbose output
```

14.3.2. Extended help

If you want to add more information to the automatically generated help message, use the `epilog` parameter when creating the `ArgumentParser` object.

```
parser = argparse.ArgumentParser(
    description='Demo',
    epilog="My extended help text"
)
```

By default, `argparse` strips all leading/trailing whitespace from the `epilog` string and displays it as a single long line. If you want to preserve your formatting (including newlines and indentation), use the `RawDescriptionHelpFormatter`.

```
parser = argparse.ArgumentParser(
    description='Demo',
    formatter_class=argparse.RawDescriptionHelpFormatter,
    epilog="""
My longer text
    includes fancy
             whitespace formatting
    """
    )
```

14.3.3. Parameter error checking

A useful side effect of using `argparse` is that it automatically generates an error if a user passes in an unrecognized command-line argument. This includes unknown flags or extra positional arguments.

```
$ python test.py filename
usage: test.py [-h] [--verbose]
test.py: error: unrecognized arguments: filename
```

14.3.4. Multiple, Short or Long Flags

For short and long versions of a flag, such as `-v` and `--verbose`, specify multiple flag like so:

```
import argparse

parser = argparse.ArgumentParser()
parser.add_argument('--verbose', '-v', action='store_true')

args = parser.parse_args()

if args.verbose:
    print("~ Verbose!")
else:
    print("~ Not so verbose")
```

14.3.5. Required Flags

Make a flag mandatory by setting `required=True`. If the flag is not specified when the script is run, `argparse` will raise an error.

```
parser = argparse.ArgumentParser()
parser.add_argument('--limit', required=True, type=int)
args = parser.parse_args()
# Access args.limit here
```

14.4. Positional Arguments

The examples so far have focused on flags (parameters starting with `--`). However, `argparse` also handles positional arguments, which are specified without a preceding flag. Here's an example to illustrate:

```
import argparse

parser = argparse.ArgumentParser()
parser.add_argument('filename')
args = parser.parse_args()

print(f"~ Filename: {args.filename}")
```

Output:

```
$ python test.py filename.txt
~ Filename: filename.txt
```

14.5. Number of Arguments

Argparse determines the number of arguments expected based on the `action` specified. For instance, in the verbose example, the `store_true` action takes no argument. By default, if no `action` implies otherwise, `argparse` expects a single argument for a positional argument or a flag that takes a value, as shown in the `filename` example above.

If you want a parameter to accept a specific number of items, specify `nargs=N`, where `N` is the number of arguments to accept. Note: if you set `nargs=1`, the argument will be stored as a list containing a single value, not as the value itself.

```
import argparse

parser = argparse.ArgumentParser()
parser.add_argument('nums', nargs=2)
args = parser.parse_args()
```

```
print(f"~ Nums: {args.nums}")
```

Output:

```
$ python test.py 5 2
~ Nums: ['5', '2']
```

14.5.1. Variable Number of Parameters

The `nargs` parameter also accepts special string values to handle a variable number of arguments:

- `*` (asterisk): Collects all command-line arguments present into a list. If no arguments are provided for this parameter, it will be an empty list.

```
import argparse

parser = argparse.ArgumentParser()
parser.add_argument('nums', nargs='*')
args = parser.parse_args()

print(f"~ Nums: {args.nums}")
```

Output:

```
$ python test.py 5 2 4
~ Nums: ['5', '2', '4']

$ python test.py
~ Nums: []
```

- `+` (plus sign): Similar to `*`, but requires at least one argument. If no arguments are provided, `argparse` will raise an error.

```
import argparse

parser = argparse.ArgumentParser()
parser.add_argument('files', nargs='+', help='One or more files to process
')
args = parser.parse_args()

print(f"~ Files: {args.files}")
```

Output:

```
$ python test.py file1.txt file2.txt
~ Files: ['file1.txt', 'file2.txt']

$ python test.py
usage: test.py [-h] files [files ...]
test.py: error: the following arguments are required: files
```

Positional arguments are consumed in the order they are defined. You can combine `nargs='*'` with other positional arguments. For example, to define a required filename followed by a list of optional values:

```
import argparse

parser = argparse.ArgumentParser()
parser.add_argument('filename')
parser.add_argument('nums', nargs='*')
args = parser.parse_args()

print(f"~ Filename: {args.filename}")
print(f"~ Nums: {args.nums}")
```

Output:

```
$ python test.py file.txt 5 2 4
~ Filename: file.txt
~ Nums: ['5', '2', '4']
```

- `?` (question mark): Makes a positional argument optional. If the argument is present, it's consumed; if not, the `default` value is used (which is `None` if not otherwise specified by the `default` parameter in `add_argument`). Be careful when combining `?` and for positional arguments, especially if an optional positional argument (`nargs='?'`) precedes one that accepts multiple arguments (`nargs=''`).

This example demonstrates a common use case: a required filename and an optional second argument:

```
import argparse

parser = argparse.ArgumentParser()
parser.add_argument('filename') # Required
parser.add_argument('output_filename', nargs='?') # Optional
args = parser.parse_args()

print(f"~ Input: {args.filename}")
print(f"~ Output: {args.output_filename}")
```

Output:

```
$ python test.py input.txt output.txt
~ Input: input.txt
~ Output: output.txt

$ python test.py input.txt
~ Input: input.txt
~ Output: None
```

However, if you define an optional argument (`nargs='?'`) before a variable-length argument (`nargs='*'`), and then provide only one argument on the command line, `argparse` will assign it to the first (optional) argument:

```
import argparse

parser = argparse.ArgumentParser()
parser.add_argument('optional_file', nargs='?', default='default.txt')
parser.add_argument('other_args', nargs='*')
args = parser.parse_args()

print(f"~ Optional File: {args.optional_file}")
print(f"~ Other Args: {args.other_args}")
```

Output:

```
$ python test.py file1.txt arg2 arg3
~ Optional File: file1.txt
~ Other Args: ['arg2', 'arg3']

$ python test.py only_one_arg
~ Optional File: only_one_arg
~ Other Args: []

$ python test.py
~ Optional File: default.txt
~ Other Args: []
```

You can also use `nargs` with flag arguments (option arguments that start with `-` or `--`).

```
import argparse

parser = argparse.ArgumentParser()
parser.add_argument('--point', nargs=2, help='Specify X and Y coordinates')
parser.add_argument('--tags', nargs='+', help='Specify one or more tags')
```

```
parser.add_argument('object_type') # Positional argument
args = parser.parse_args()

if args.point:
    print(f"~ Point: ({args.point[0]}, {args.point[1]})")
if args.tags:
    print(f"~ Tags: {args.tags}")
print(f"~ Type: {args.object_type}")
```

Output:

```
$ python test.py --point 5 10 --tags foo bar baz square
~ Point: (5, 10)
~ Tags: ['foo', 'bar', 'baz']
~ Type: square

$ python test.py --tags urgent important circle
~ Tags: ['urgent', 'important']
~ Type: circle
```

14.6. Variable Type

You might have noticed in previous examples that command-line arguments are typically treated as strings. To have `argparse` convert an argument to a specific data type, use the `type` parameter. For example, to convert arguments to integers, specify `type=int`. If an argument cannot be converted to the specified type, `argparse` will automatically raise an error.

```
import argparse

parser = argparse.ArgumentParser()
parser.add_argument('nums', nargs=2, type=int)
args = parser.parse_args()

print(f"~ Nums: {args.nums}")
print(f"~ Type of first num: {type(args.nums[0])}")
```

Output:

```
$ python test.py 5 2
~ Nums: [5, 2]
~ Type of first num: <class 'int'>

$ python test.py 5 foo
usage: test.py [-h] nums nums
test.py: error: argument nums: invalid int value: 'foo'
```

14.6.1. File Types

Argparse provides built-in `FileType` objects that can open files specified on the command line. You can use `argparse.FileType('r')` for reading, `argparse.FileType('w')` for writing, `argparse.FileType('a')` for appending, and modes like `'rb'` or `'wb'` for binary files. `argparse` will attempt to open the file, and will raise an error if it fails to open.

I don't recommend using `FileType` objects in argparse, it requires remembering to do something (closing the file) and that's always a bad idea to force yourself to have to remember something. I prefer being explicit and use the `open` function with a context manager (`with`) that automatically closes the file. See Files for more.

14.7. Default Value

You can specify a default value for an argument if the user does not provide one. Use the `default` parameter in `add_argument`. This is especially useful for optional arguments (flags or positional arguments with `nargs='?'`).

Here's an example using a flag:

```
import argparse

parser = argparse.ArgumentParser()
parser.add_argument('--limit', default=5, type=int)
args = parser.parse_args()

print(f"~ Limit: {args.limit}")
```

Output:

```
$ python test.py
~ Limit: 5

$ python test.py --limit 10
~ Limit: 10
```

14.8. Remainder

If you want to gather all arguments that were not parsed by other argument specifications, you can use `argparse.REMAINDER`. This will collect all remaining command-line arguments into a list. This is useful when you want to pass arguments through to another script or command.

```
import argparse

parser = argparse.ArgumentParser()
parser.add_argument('--verbose', action='store_true', help='verbose flag')

# All arguments after --verbose will be captured by 'command_args'.
```

```
parser.add_argument('command_args', nargs=argparse.REMAINDER)
args = parser.parse_args()

if args.verbose:
    print("Verbose mode is on.")
print(f"Command and its arguments: {args.command_args}")
```

Specifying `argparse.REMAINDER` will create a list of all unparsed arguments:

```
$ python test.py --verbose cmd --option1 value1 arg1
Verbose mode is on.
Command and its arguments: ['cmd', '--option1', 'value1', 'arg1']

$ python test.py other_cmd --flag
Command and its arguments: ['other_cmd', '--flag']
```

14.9. Actions

The `action` parameter in `add_argument()` determines how the command-line argument is processed and stored. The default action is `'store'`, which simply stores the argument's value (or `None` if it's a flag that isn't present and has no default). You've already seen `'store_true'`. Let's explore some other useful actions.

14.9.1. Booleans

You have already seen `action='store_true'`, which sets the argument to `True` if the flag is present, and `False` otherwise (or its `default` value if one is provided and the flag isn't present, though this is less common for boolean flags where the absence implies `False`).

Its counterpart is `action='store_false'`. This sets the argument to `False` if the flag is present and `True` otherwise. This can be useful for flags that disable a default behavior.

```
import argparse

parser = argparse.ArgumentParser()
parser.add_argument('--no-feature', action='store_false', dest=
'feature_enabled')
parser.set_defaults(feature_enabled=True) # Feature is enabled by default
args = parser.parse_args()

if args.feature_enabled:
    print("Feature is ON")
else:
    print("Feature is OFF (disabled by --no-feature)")
```

Output:

```
$ python test.py
Feature is ON

$ python test.py --no-feature
Feature is OFF (disabled by --no-feature)
```

14.9.2. Count

Use `action='count'` to count the number of times a flag appears. This is often used for increasing verbosity levels (e.g., `-v`, `-vv`, `-vvv`). If the flag is not present, the value will be `None` unless a `default` is set.

```
import argparse

parser = argparse.ArgumentParser()
parser.add_argument('--verbose', '-v', action='count', default=0) # Default
to 0 if not specified
args = parser.parse_args()

if args.verbose == 0:
    print("Not verbose.")
elif args.verbose == 1:
    print("A little verbose.")
elif args.verbose == 2:
    print("More verbose!")
else:
    print(f"Verbosity level: {args.verbose}")
```

Output:

```
$ python test.py
Not verbose.

$ python test.py -v
A little verbose.

$ python test.py --verbose -v --verbose
Verbosity level: 3
```

14.9.3. Append

Use `action='append'` to store a list of values. Each time the flag is encountered, the provided value is appended to the list. If the flag is not used, the value will be `None` unless a `default` is specified.

```
import argparse

parser = argparse.ArgumentParser()
parser.add_argument('-I', '--include-dir', action='append', dest=
'include_dirs', help='Add directory to include path')
parser.add_argument('--other', action='append') # Will be stored in
args.other
args = parser.parse_args()

print(f"~ Include Directories: {args.include_dirs}")
print(f"~ Other appended args: {args.other}")
```

Output:

```
$ python test.py
~ Include Directories: None
~ Other appended args: None

$ python test.py -I /usr/local/include --other val1
~ Include Directories: ['/usr/local/include']
~ Other appended args: ['val1']

$ python test.py -I ./include -I ../headers --other itemA --other itemB
~ Include Directories: ['./include', '../headers']
~ Other appended args: ['itemA', 'itemB']
```

14.10. Choices

If you want to restrict an argument to a specific set of allowed values, use the `choices` parameter. Provide a list of valid options. If the user enters a value not in this list, `argparse` will display an error message. The `choices` are usually displayed in the help message.

```
import argparse

parser = argparse.ArgumentParser(prog='game.py')
parser.add_argument('move', choices=['rock', 'paper', 'scissors'], help=
'Your throw in rock-paper-scissors')
args = parser.parse_args()

print(f"~ Your move: {args.move}")
```

Output (success):

```
$ python game.py rock
```

```
~ Your move: rock
```

Output (error):

```
$ python game.py lizard
usage: game.py [-h] {rock,paper,scissors}
game.py: error: argument move: invalid choice: 'lizard' (choose from
'rock', 'paper', 'scissors')
```

14.11. Examples

I'll end with two more complete examples. Many of the examples above were kept short to focus on the specific idea being illustrated. These show how you might combine several `argparse` features in a script.

14.11.1. Copy Script Example

Here is a script that copies one file to another, with options for verbosity and recursive copying (though the recursive logic itself is not implemented here, only the argument parsing).

```
import argparse

# Create the parser
parser = argparse.ArgumentParser(description='A script to copy one file to
another.')

# Add arguments
parser.add_argument('-v', '--verbose',
    action="store_true",
    help="Enable verbose output." )

parser.add_argument('-R', '--recursive',
    action="store_true",
    help="Copy all files and directories recursively" )

parser.add_argument('infile', help="The source file to be copied.")

parser.add_argument('outfile',help="The destination file")

# Parse arguments
args = parser.parse_args()

# --- Main script logic would go here ---
if args.verbose:
    print(f"Verbose mode enabled.")
    print(f"Copy from: {args.infile}")
    print(f"Copy to: {args.outfile}")
```

```
    if args.recursive:
        print("Recursive copy requested")

# -- implement logic

if args.verbose:
    print("Copy complete.")

print(f"Successfully copied '{args.infile}' to '{args.outfile}'.")
```

Example running:

```
$ python copy_script.py source.txt dest.txt
Successfully copied 'source.txt' to 'dest.txt'.

$ python copy_script.py -v source.txt dest.txt
Verbose mode enabled.
Copy from: source.txt
Copy to: destination_verbose.txt
Copy complete.
Successfully copied 'source.txt' to 'dest.txt'.

$ python copy_script.py -v -R source.txt dest.txt
Verbose mode enabled.
Copy from: source.txt
Copy to: dest.txt
Recursive copy requested
Copy complete.
Successfully copied 'source.txt' to 'dest.txt'.
```

14.11.2. Bug Script Example

Here is an example of a script that might be used to update a bug's status in a bug tracking system.

```
import argparse

parser = argparse.ArgumentParser(description='Update the status of a bug.')

parser.add_argument('-v', '--verbose',
    action="store_true",
    help="Enable verbose output." )

parser.add_argument('-s', '--status',
    default="closed",
    choices=['closed', 'wontfix', 'notabug', 'reopened', 'inprogress'],
    help="Set the bug status. Defaults to \'closed\'.")
```

```
parser.add_argument('bug_id',
    type=int,
    help="The numeric ID of the bug to update.")

parser.add_argument('message',
    nargs='*',
    help="An optional message to record with the status update.")

args = parser.parse_args()

# --- Main script logic (e.g., interacting with a bug system API) ---

if args.verbose:
    print("Verbose mode enabled.")
    print(f"Attempting to update bug ID: {args.bug_id}")
    print(f"New status: {args.status}")
    if args.message:
        print(f"Message: {' '.join(args.message)}")

# Simulate updating the bug
print(f"Bug {args.bug_id} status set to '{args.status}'.")
if args.message:
    full_message = " ".join(args.message)
    print(f"Message recorded: \"{full_message}\"")
else:
    print("No message recorded.")

if args.verbose:
    print("Bug update process complete.")
```

Example usage:

```
$ python bug_script.py 12345
Bug 12345 status set to 'closed'.
No message recorded.

$ python bug_script.py -s wontfix 67890 This is an old issue, not relevant
anymore.
Bug 67890 status set to 'wontfix'.
Message recorded: "This is an old issue, not relevant anymore."

$ python bug_script.py -v --status=inprogress 54321 Working on the fix now
Verbose mode enabled.
Attempting to update bug ID: 54321
New status: inprogress
Message: Working on the fix now
Bug 54321 status set to 'inprogress'.
```

```
Message recorded: "Working on the fix now"
Bug update process complete.
```

Chapter 15. Dates and Time

A set of examples working with Python date and time functions, including formatting dates, date calculations, and other common date tasks.

First off, all examples use the following import, any additional imports needed will be shown with the example.

```
from datetime import datetime
```

15.1. Creating Date Objects

To start, you'll usually want to create a Python date object, either from the current time, a specific known time, or by parsing a string.

15.1.1. Create a Specific Date

```
# now
now = datetime.now()

# specific date
dt1 = datetime(2011, 8, 29)
dt2 = datetime(year=2012, month=3, day=2)
```

15.1.2. Create Date from Timestamp

Create a Python date object from a unix timestamp, a 10-digit number counting seconds from the epoch

```
ts = 1294204471
dt = datetime.fromtimestamp(ts)
```

15.1.3. Create Date from Known Format

To create a Python date object from a string with a known format, use `strptime()`. See the format table below for definitions.

```
str = "2012-10-20"
dts = datetime.strptime(str, '%Y-%m-%d')
```

15.1.4. Create Date from Unknown Format

To create a Python date object from a string with an **unknown** date format, it's best to use the third-party dateutil module. Install using: `python -m pip install python-dateutil`

```
import dateutil.parser as parser

str1 = "October 11, 2018"
str2 = "Oct 11, 2018"
str3 = "Oct 11th, 2018"
parser.parse(str1)
parser.parse(str2)
parser.parse(str3)

# datetime.datetime(2018, 10, 11, 0, 0)
```

15.2. Date Format Table

Printing dates in various formats is relatively straightforward using `strftime()`. Here's one example.

```
dt = datetime(2019, 8, 22)
dt.strftime("%b %d, %Y")
>>> 'Aug 22, 2019'
```

The table below shows ther available formatting symbols and samples.

Symbol	Definition	Example
%a	Weekday name abbreviated	Sun, Mon, Tue, …
%A	Weekday name full	Sunday, Monday, Tuesday, …
%b	Month name abbreviated	Jan, Feb, Mar, …
%B	Month name full	January, February, …
%c	A "random" date and time representation.	Fri Jan 15 16:34:00 1999
%d	Day of the month	[01, 31]
%f	Microsecond	[000000, 999999]
%H	Hour (24h)	[00, 23]
%I	Hour (12h)	[01, 12]
%j	Day of the year	[001, 366]
%m	Month	[01, 12]
%M	Minute	[00, 59]
%p	Locale's equivalent of either AM or PM.	[AM, PM]
%S	Second	[00, 61]
%U	Week number of the year (Sunday first)	[00, 53]
%w	Weekday number (Sunday=0)	[0, 6]
%W	Week number of the year (Monday first)	[0, 53]
%x	Locale date	01/15/99
%X	Locale time	16:34:00
%y	Year without century	[00, 99]
%Y	Year with century	1999
%z	UTC offset in the form +HHMM or -HHMM or empty	
%Z	Time zone name or empty string	
%%	A literal '%' character.	

15.3. Format Date to RFC 2822

Use the `formatdate()` function from the `email.utils` built-in module.

For the current date and time:

```
from email.utils import formatdate
formatdate()
>>> 'Sat, 04 Mar 2023 04:36:44 -0000'
```

If you need to format a specific date, pass in a unix timestamp.

```
from email.utils import formatdate
dt = datetime(2020, 1, 25)
formatdate(dt.timestamp())
>>> 'Sat, 25 Jan 2020 08:00:00 -0000'
```

15.4. Date Calculations and Timedelta

```
from datetime import timedelta

week_later = dt + timedelta(days=7)
last_week = dt - timedelta(days=7)
in_five_minutes = dt + timedelta(minutes=5)
```

Valid timedelta properties are: `weeks`, `days`, `hours`, `minutes`, `seconds`, `microseconds`, `milliseconds`

You might notice that a "year" timedelta is absent. Don't be tempted to use `days=365`, as this would be incorrect for leap years. I recommend something like the following:

```
st = datetime(year=2011, month=3, day=17)
next_year = datetime(year=st.year+1, month=st.month, day=st.day)
```

15.4.1. Adding and Subtracting Dates

You can add and subtract date objects. Doing so returns `timedelta` objects. Using the `timedelta` object, you can access the same properties listed above.

```
dt1 = datetime(year=2012, month=8, day=23)
dt2 = datetime(year=2012, month=8, day=28)
td = dt2 - dt1
td.days
```

```
>>> 5
```

15.5. Convert Datetime to Date

To simplify a `datetime` object to just the date, convert it to a `date` object using the `.date()` method:

```
dt = datetime.now()
dt
>>> datetime.datetime(2024, 3, 19, 5, 46, 17, 591196)
dt.date()
>>> datetime.date(2024, 3, 19)
```

15.6. Convert Date to Datetime

To add a time to a `date` object (converting it to a `datetime` object), use the `.combine()` method. This merges a `date` object with a `time` object. Be careful with your imports to ensure you use the `datetime.time` object.

```
from datetime import date
from datetime import datetime
from datetime import time as dt_time

dt = datetime.combine(date.today(), dt_time(0,0,0,0))
```

15.7. Common Date Operations

Here are examples of some common Python date operations.

15.7.1. Yesterday

To get yesterday's date as a Python `date` object, use `timedelta`:

```
from datetime import date, timedelta
yesterday = date.today() - timedelta(days=1)
yesterday.strftime('%Y-%m-%d')
>>> '2020-02-08'
```

15.7.2. Last Day of the Month

Here are two solutions. First, using `datetime`, go to the first day of the next month and subtract one day.

```
from datetime import datetime, timedelta
```

```
now = datetime.now()
# Handle December correctly by adjusting year and month
if now.month == 12:
    next_month_year = now.year + 1
    next_month_month = 1
else:
    next_month_year = now.year
    next_month_month = now.month + 1

next_month_first_day = datetime(year=next_month_year, month
=next_month_month, day=1)
last_day_month = next_month_first_day - timedelta(days=1)
```

Second solution to determine the last day of the month using the `calendar` module.

```
import calendar
from datetime import datetime
now = datetime.now()
month_info = calendar.monthrange(now.year, now.month) # Renamed 'range' to
'month_info'
last_day_month = now.replace(day=month_info[1])
```

15.7.3. Next Thursday

To find the date of the next Thursday, note: in Python for day of week, Monday is 0 and Sunday is 6, so Thursday is 3.

```
from datetime import datetime, timedelta

today = datetime.now()
thursday_dow = 3
today_dow = today.weekday()
adjustment = (thursday_dow - today_dow + 7) % 7

if adjustment == 0: # get next Thursday
    adjustment = 7
next_thursday = today + timedelta(days=adjustment)
```

15.7.4. First Monday of the Month

To find the first Monday of the current month:

```
from datetime import datetime, timedelta # Ensure necessary imports

today = datetime.now()
```

```
first_day_of_month = today.replace(day=1)
# strftime("%w") returns Sunday as 0, Monday as 1, ..., Saturday as 6
monday_val = 1 # Monday is 1 with %w
day_of_week = int(first_day_of_month.strftime("%w")) # Convert to int

# Calculate days to add to reach the first Monday
if day_of_week == monday_val:
    adjustment = 0 # First day is already Monday
elif day_of_week < monday_val: # e.g. Sunday (0)
    adjustment = monday_val - day_of_week
else: # e.g. Tuesday (2) through Saturday (6)
    adjustment = 7 - day_of_week + monday_val

first_monday = first_day_of_month + timedelta(days=adjustment)
```

Chapter 16. Regular Expressions

Regular expressions in Python use the standard module re.

Use raw strings for a regex pattern to avoid issues with escaping special characters. Recent versions of Python will issue a warning when not using a raw string for escaped characters.

16.1. Basic regular expression matching

The two main regex functions are `re.search()` and of `re.match()`. The difference is that `re.search()` scans the entire string to find a match for the expression anywhere, while `re.match()` only checks for a match at the **beginning** of the string. The majority of the time you'll want to just use `re.search()`.

```
import re

s = "Hello world"
m = re.match("Hello", s)
if m:
    print("Match found.")
else:
    print("No match.")

>>> Match found.
```

Trying to match "world" highlights the difference between `.search()` and `.match()`:

```
import re

s = "Hello world"
m = re.match("world", s)
if m:
    print("Match found.")
else:
    print("No match.")

>>> No match.


m = re.search("world", s)
if m:
    print("Match found.")
else:
    print("No match.")

>>> Match found.
```

16.2. Capturing Matches

Often times you don't want to just match a pattern, but capture a value. Use parentheses `()` to group parts of a pattern and capture the value:

```
import re

m = re.search(r"(\d+)", s)
if m is not None:
    print(f"Search found: {m.group(1)} dogs")
```

Both `re.match()` and `re.search()` return a Match object if a match is found, or `None` otherwise. Use `m.group()` to get the matched substring, `m.group(1)` returns the substring matched by the first group, `m.group(2)` by the second, and so on.

Here's an example with multiple capturing groups:

```
import re

s = "There are 13 dogs outside and 2 cats"

m = re.search(r"(\d+).*(\d+)", s)
if m is not None:
    print(f"Matched String: {m.group(0)}")
    print(f"First : {m.group(1)}")
    print(f"Second: {m.group(2)}")

>>> Matched String: 13 dogs outside and 2
>>> First : 13
>>> Second: 2
```

16.3. Find All Matches

Use `re.findadll(<pattern>, string)` to return a list of all substrings that match pattern within a string.

```
s = "There are 13 dogs outside and 2 cats"
m = re.findall(r"\d+", s)
print(m)
>>> ['13', '2']
```

☞ What do you think will match using `re.findall(r"\d", s)` without the `+` ?

16.4. Regular Expression Substitution

Use the `re.sub()` function to replace text based on a regular expression. `re.sub()` will replace all occurrences of the pattern unless a `count` argument is specified. It returns a new string with the replacements made.

```
import re

s = "There are 13 dogs outside and 2 cats."

# Replace all digits with "many"
s_replaced = re.sub(r"\d+", "many", s)
print(s_replaced)
>>> There are many dogs outside and many cats

# Replace only the first occurrence
s_replaced_once = re.sub(r"\d+", "several", s, count=1)
print(s_replaced_once)
>>> There are several dogs outside and 2 cats.
```

16.5. Split Strings Using Regular Expressions

Use `re.split()` to split a string based on regular expressions.

```
import re

s = "apple,orange;banana:grape"
fruits = re.split(r"[,;:]", s)
print(fruits)

>>> ['apple', ' orange', 'banana', 'grape']
```

16.6. Regular Expression Patterns

The examples have used the `\d+` command to match 1 or more digits. Here are the other standard regular expression patterns that Python supports.

Pattern	Description
`.`	Any character except newline
`\d`	Digits (0-9)
`\w`	Word character (letters, digits, underscore)
`\s`	Whitespace (spaces, tabs, newlines)
`^`	Start of string
`$`	End of string
`*`	0 or more
`+`	1 or more
`?`	0 or 1
`{n}`	Exactly n times
`a|b`	Use pipe to match a or b

16.6.1. Characters

Examples matching against digit and word characters.

```
text = "User 42 logged in at 10:45"
digits = re.findall(r"\d", text)
print(digits)
>>> ['4', '2', '1' '0', '4', '5']

text = "User 42 logged in at 10:45"
digits = re.findall(r"\w+", text)
print(digits)
>>> ['User', '42', 'logged', 'in', 'at', '10', '45']
```

16.6.2. Anchors

Examples matching against anchor characters, start and end of strings.

```
text = "python is fun"
if re.match(r"^python", text):
    print("Starts with python")

if re.search(r"fun$", text):
    print("Ends with fun")
```

16.6.3. Quantifiers

Examples using quanitifiers, matching aginst certain numbers of characters.

```
text = "aaaab"
m = re.match(r"a{4}b", text) # match 4 a's
print(m.group())
>>> aaaab
```

The `.` is a special character to match anything, using `.*` would always return true, since asterisk means match 0 or more. Here's an example matching an HTML header that might contain various characters.

```
html = "<h1>My Page Title</h1>"
m = re.search(r"<h1>(.*)</h1>", html)
if match:
    print(m.group())
    print(m.group(1))
>>> <h1>My Page Title</h1>
>>> My Page Title
```

Use `|` to match either pattern:

```
text = "I like cats and dogs"
match = re.findall(r"cats|dogs", text)
print(match)
>>> ['cats', 'dogs']
```

Chapter 17. System Commands

To embrace the unix way, letting each tool do its thing, you need to be able to execute a system command from Python. There are a few ways to call shell commands, but the modern and recommended way is with the `subprocess` module.

17.1. `os.system()`

An older, simpler way to run external commands is by using `os.system()`. This function is best suited for straightforward, "one-liner" tasks where you only need to execute a command and don't need to capture its output or perform complex error handling.

```
import os
exit_code = os.system('ls -l')
print("Exit code:", exit_code)
```

An exit code of 0 typically indicates success.

Downsides: You only receive the exit code from the command, not its output (stdout or stderr). For more control and to access the command's output, you should use the `subprocess` module.

17.2. `subprocess.run()`

For a more robust and flexible approach to running system commands, use the `subprocess.run()` function. It offers better control over command execution and allows you to access the results. You provide the command and its arguments as a list of strings.

17.2.1. Basic Example

```
import subprocess
subprocess.run(["ls", "-l"])
# prints directory listing
```

17.2.2. Capture Command Output

By default, `subprocess.run()` executes the command and displays its output directly in your terminal, just as if you had run it yourself. To capture this output into a Python string, set the `capture_output=True` parameter. Additionally, use `text=True` to automatically decode the command's output from bytes to a string using your system's default encoding (usually UTF-8).

```
import subprocess
result = subprocess.run(["ls", "-l"], capture_output=True, text=True)
print(result.stdout)
```

17.2.3. Handling Timeouts

Sometimes, an external command might hang indefinitely. To prevent your Python script from also hanging, you can use the `timeout` parameter (in seconds). If the command exceeds this timeout, a `subprocess.TimeoutExpired` exception is raised.

```
import subprocess
try:
    result = subprocess.run(["sleep", "10"], timeout=5, capture_output=
True, text=True)
    print("Command finished within timeout")
except subprocess.TimeoutExpired:
    print("Command timed out!")
except subprocess.CalledProcessError as e:
    print(f"Command failed: {e}")
```

17.2.4. Understanding Encodings

When `text=True`, `subprocess` uses the system's default encoding to decode output from commands. Usually, this is UTF-8 and works fine. However, if a command outputs text in a different encoding, you might see garbled characters or errors. In such cases, you can specify the encoding explicitly:

```
# Example assuming a command outputs Latin-1 encoded text
result = subprocess.run(["some_command_with_latin1_output"],
capture_output=True, text=True, encoding='latin-1')
print(result.stdout)
```

If you are dealing with binary data, or need precise control over byte streams, you should omit `text=True` (and `encoding`) and work directly with the `result.stdout` and `result.stderr` as byte strings.

17.2.5. Run a command and check for errors

External commands can fail for various reasons. You can make `subprocess.run()` automatically raise an exception for non-zero exit codes by setting `check=True`. You should then use a `try-except` block to catch the `subprocess.CalledProcessError` and handle the failure gracefully.

```
try:
    subprocess.run(["git", "status"], check=True)
except subprocess.CalledProcessError as e:
    print(f"The git command failed with exit code {e.returncode}")
    if e.stdout:
        print(f"Stdout:\n{e.stdout}")
    if e.stderr:
        print(f"Stderr:\n{e.stderr}")
```

17.3. Running with a shell

When you use `shell=True`, Python runs the command through your shell (like bash or zsh). This allows you to use shell features directly in your command string, such as: - Use pipes (e.g., `cat foo.txt | grep bar`) - Expand wildcards (e.g., `*.py`) - Use environment variables (e.g., `$HOME`)

Example:

```
subprocess.run("echo $HOME", shell=True)
# prints your home directory
```

Security Warning: Using `shell=True` with untrusted input can be a security risk due to potential shell injection vulnerabilities. If any part of the command comes from external input (e.g., user input, web request), it's highly recommended to avoid `shell=True` or to meticulously sanitize the input. Prefer passing arguments as a sequence (e.g., `['ls', '-l']`) whenever possible, as this is inherently safer.

17.4. Advanced: `subprocess.Popen`

For more advanced scenarios, such as non-blocking execution or complex I/O piping, you can use `subprocess.Popen`. It gives you finer-grained control over process creation and management, allowing you to: - Read output as it is produced (line by line) - Send input to a command while it runs - Pipe output between commands (like chaining with `|`) - Start background processes

17.4.1. Example: Stream output as command runs

```
import subprocess
proc = subprocess.Popen(["ping", "-c", "3", "example.com"], stdout
=subprocess.PIPE, text=True)
for line in proc.stdout:
    print("got:", line.rstrip())
```

17.4.2. Example: Send input to a process

```
proc = subprocess.Popen(["cat"], stdin=subprocess.PIPE, text=True)
proc.communicate(input="Hello from Python!\n")
# 'cat' will echo this right back out
```

17.4.3. Example: Pipe output between commands

```
from subprocess import Popen, PIPE
ps = Popen(["ps", "aux"], stdout=PIPE)
grep = Popen(["grep", "python"], stdin=ps.stdout, stdout=PIPE, text=True)
ps.stdout.close()  # Allow ps to get SIGPIPE if grep closes
output = grep.communicate()[0]
```

```
print(output)
```

For many common tasks, `subprocess.run()` is sufficient. It can also accept input directly via its `input` argument (e.g., `subprocess.run(['grep', 'pattern'], input='some text', text=True)`). Reserve `subprocess.Popen` for situations where you truly need its advanced capabilities, like asynchronous execution or managing multiple communicating processes.

Chapter 18. Dev Environment

18.1. Python Development with uv

I recommend using uv for Python installation and package management. It's an extremely fast Python package manager written in Rust that replaces pip, virtualenv, pyenv, and more in a single tool.

18.1.1. Installing uv

On macOS and Linux:

```
curl -LsSf https://astral.sh/uv/install.sh | sh
```

On Windows:

```
powershell -c "irm https://astral.sh/uv/install.ps1 | iex"
```

Or with Homebrew:

```
brew install uv
```

18.1.2. Installing Python

With uv, you can install and manage Python versions easily:

```
# Install the latest Python version
uv python install

# Install a specific version
uv python install 3.12

# List available and installed versions
uv python list
```

18.1.3. Creating Projects

Start a new Python project:

```
# Create a new project
uv init my-project
cd my-project

# Add dependencies
```

```
uv add requests flask

# Run your code
uv run app.py
```

For existing projects, create a virtual environment:

```
uv venv
source .venv/bin/activate  # On macOS/Linux
# or
.venv\Scripts\activate     # On Windows
```

18.2. Visual Studio Code / Cursor Setup

I recommend using Visual Studio Code as your primary Python editor, or Cursor if you want that AI goodness. Both provide excellent Python support with extensions and built-in features.

18.2.1. Essential Extensions

Install these extensions for the best Python development experience:

- **Python** - Official Python extension with IntelliSense, debugging, and formatting
- **Pylance** - Fast, feature-rich language server for Python
- **Ruff** - Fast Python linter and formatter

18.2.2. VS Code Settings

Here are example settings for your VS Code `settings.json` to auto format on save, using Ruff for both linting and formatting:

```
{
    "python.defaultInterpreterPath": ".venv/bin/python",
    "python.terminal.activateEnvironment": true,
    "[python]": {
        "editor.defaultFormatter": "charliermarsh.ruff",
        "editor.formatOnSave": true,
        "editor.codeActionsOnSave": {
            "source.organizeImports": "explicit"
        }
    },
    "ruff.fixAll": true,
    "ruff.organizeImports": true
}
```

18.2.3. Formatting and Linting with Ruff

Ruff is a fast Python linter and formatter written in Rust. It can handle both code formatting and linting in a single tool, making it simpler than using separate tools like Black. With the extension installed, it will automatically format your code on save and show errors and warnings inline.

You can also run ruff from the command line:

```
# Install ruff in your project
uv add --dev ruff

# Run the linter
uv run ruff check .

# Auto-fix issues
uv run ruff check --fix .
```

Chapter 19. Project Structure

Organizing your Python projects with a clear structure and modern tools makes your code easier to maintain, test, and share. Here's how I scaffold a project with a standardized layout, using `pyproject.toml` for configuration and `uv` for environment and dependency management.

19.1. Standard Project Structure

This layout separates source code, tests, documentation, and metadata:

```
myproject/
    README.md
    pyproject.toml
    src/
        myproject/
            __init__.py
            main.py
            utils.py
    tests/
        __init__.py
        test_main.py
    docs/
```

Key Points:

- `src/myproject/`: All your Python source lives here.
- `tests/`: Your test code.
- `docs/`: Documentation.
- `pyproject.toml`: Project and dependency configuration.
- `README.md`: Project description.

19.2. Why pyproject.toml and uv?

`pyproject.toml` is the new standard for Python project configuration. It unifies settings for packaging, dependencies, formatting, and more. See the official documentation at Python packaging docs.

As covered in the Dev Environment chapter, `uv` is a fast Python package manager and virtual environment tool, compatible with PEP 508 dependency specification and `pyproject.toml`. It's a drop-in replacement for pip and virtualenv and simplifies managing a Python project.

19.2.1. Example pyproject.toml

```
[project]
name = "myproject"
version = "0.1.0"
```

```
description = "A sample Python project"
readme = "README.md"
requires-python = ">=3.10"

[project.dependencies]
rich = "^13.0.0"

[tool.pytest.ini_options]
pythonpath = ["src"]
addopts = "-ra"
```

Add any extra dependencies you need under `[project.dependencies]`.

19.2.2. Using uv for a local environment

In your project directory, create a per project virtual environment:

```
uv venv
source .venv/bin/activate
```

Install dependencies from `pyproject.toml`:

```
uv sync
```

To add a new dependency, `uv add` will add to `pyproject.toml` and sync:

```
uv add requests
```

Use `uv remove` to remove a dependency:

```
uv remove requests
```

19.2.3. Run Your Code

For any new terminal session, be sure to activate your virtual environment (see Tips below):

```
source .venv/bin/activate
```

To run your code:

```
python src/myproject/main.py
```

Test Your Code

To run tests, first add `pytest` as a dev dependency:

```
uv add --dev pytest
```

This will add a section to `pyproject.toml`:

```
[dependency-groups]
dev = [
    "pytest>=8.1",
]
```

To install dev dependencies use:

```
uv sync --dev
```

Run your tests using `pytest` that is now in your virtual environment:

```
pytest
```

19.3. Tips

- Use `uv` everywhere you used to use `pip` or `virtualenv` — it's much faster and uses your `pyproject.toml` directly.
- Keep dependencies declared in `pyproject.toml` for reproducibility.
- Check in your `pyproject.toml`; never commit your `.venv`.

I use Starship.rs to customize my shell to display the current virtual environment in the command prompt, this lets me know if I've activated the correct virtual environment. See the Python section of the Starship.rs documentation for details.

Chapter 20. Testing

Testing is one of those things I wish I'd learned earlier in my Python journey. It saves time debugging and gives you confidence when making changes. Here's how to get started with pytest, one of the most popular testing frameworks.

20.1. Installing pytest

Install `pytest` using `uv` or with `pip`, see Dev Environment for more on using `uv`.

```
# Using uv (recommended)
uv add pytest

# Or with pip
pip install pytest
```

That's it, `pytest` is ready to go.

20.2. Your First Test

Let's start with something simple. I'll create a basic function and then test it.

Create a file called `calculator.py`:

```
def add(a, b):
    return a + b

def multiply(a, b):
    return a * b

def divide(a, b):
    if b == 0:
        raise ValueError("Cannot divide by zero")
    return a / b
```

Now create a test file called `test_calculator.py`:

```
from calculator import add, multiply, divide
import pytest

def test_add():
    assert add(2, 3) == 5
    assert add(-1, 1) == 0
    assert add(0, 0) == 0

def test_multiply():
    assert multiply(3, 4) == 12
```

```
    assert multiply(-2, 5) == -10
    assert multiply(0, 100) == 0

def test_divide():
    assert divide(10, 2) == 5
    assert divide(7, 2) == 3.5

def test_divide_by_zero():
    with pytest.raises(ValueError):
        divide(10, 0)
```

Run the tests:

```
pytest test_calculator.py
```

You'll see output like this:

```
========================== test session starts ==========================
collected 4 items

test_calculator.py ....                                          [100%]

========================== 4 passed in 0.02s ===========================
```

20.3. Understanding Assertions

The `assert` statement is your main tool for testing. It lets you verify that your code works as expected by checking if a condition is True. If the condition is False, the test fails and raises an AssertionError. Think of it as saying "I assert that this should be True" - if it's not, something's wrong with your code.

```
def test_assertions():
    # Basic equality
    assert 2 + 2 == 4

    # Not equal
    assert 5 != 3

    # Greater than, less than
    assert 10 > 5
    assert 3 < 7

    # In/not in
    assert 'hello' in 'hello world'
    assert 'x' not in 'hello'
```

```
    # True/False
    assert True
    assert not False

    # None checks
    result = None
    assert result is None

    # Type checks
    assert isinstance([1, 2, 3], list)

    # Length checks
    assert len([1, 2, 3]) == 3
```

20.4. Testing Exceptions

When you expect your code to raise an exception, use `pytest.raises()`:

```
import pytest

def validate_age(age):
    if age < 0:
        raise ValueError("Age cannot be negative")
    if age > 150:
        raise ValueError("Age seems unrealistic")
    return age

def test_age_validation():
    # These should work fine
    assert validate_age(25) == 25
    assert validate_age(0) == 0

    # These should raise exceptions
    with pytest.raises(ValueError):
        validate_age(-1)

    with pytest.raises(ValueError):
        validate_age(200)

    # You can also check the exception message
    with pytest.raises(ValueError, match="Age cannot be negative"):
        validate_age(-5)
```

20.5. Running Tests

Here are the different ways to run tests:

```
# Run all tests
pytest

# Run tests in a specific file
pytest test_calculator.py

# Run a specific test function
pytest test_calculator.py::test_add

# Run tests with more verbose output
pytest -v

# Run tests and stop at first failure
pytest -x
```

20.6. Fixtures

Fixtures in `pytest` are a powerful feature used to set up and tear down resources needed by your tests. They are helper functions that run before your test setting up your environment or data needed for the test.

Use by decorating the helper function with `@pytest.fixture` and then passing it in as a parameter to your test. The return value from the function is set to the function variable.

```
import pytest

@pytest.fixture
def sample_data():
    return [1, 2, 3, 4, 5]

@pytest.fixture
def user_data():
    return {
        'name': 'John Doe',
        'email': 'john@example.com',
        'age': 30
    }

def test_list_operations(sample_data):
    assert len(sample_data) == 5
    assert sum(sample_data) == 15
    assert max(sample_data) == 5

def test_user_validation(user_data):
    assert user_data['name'] == 'John Doe'
    assert '@' in user_data['email']
    assert user_data['age'] > 0
```

Use `yield` to return a value from setup, and then any code after `yield` will be run in teardown. Here's an example, setting up a database connection or temporary files. Here is an example using `yield` to do both for testing a sqlite database from my tasks cli app.

```
import pytest

@pytest.fixture
def temp_file():
    """Fixture to create temp file"""
    with tempfile.NamedTemporaryFile() as tf:
        yield tf.name

    # tempfile will close and auto delete

def connection(temp_file):
    """Fixture to create temp db and yield connection"""
    conn = sqlite3.connection(temp_file)
    create_schema(conn)
    yield conn
    conn.close()

def test_task_create(connection):
    entry = { "task_entry": "Test create task" }
    task_id = Task.create(connection, entry)
    assert task_id is not None
```

20.7. Parametrized Tests

When you want to test the same function with different inputs, use `@pytest.mark.parametrize`:

```
import pytest

def is_even(n):
    return n % 2 == 0

@pytest.mark.parametrize("number,expected", [
    (2, True),
    (3, False),
    (4, True),
    (5, False),
    (0, True),
    (-2, True),
    (-3, False),
])
def test_is_even(number, expected):
    assert is_even(number) == expected
```

```
@pytest.mark.parametrize("a,b,expected", [
    (1, 2, 3),
    (0, 0, 0),
    (-1, 1, 0),
    (10, -5, 5),
])
def test_addition(a, b, expected):
    assert add(a, b) == expected
```

20.8. Testing with Files

When testing code that works with files use temporary files:

```
import tempfile
import os

def save_data(filename, data):
    with open(filename, 'w') as f:
        for item in data:
            f.write(f"{item}\n")

def load_data(filename):
    with open(filename, 'r') as f:
        return [line.strip() for line in f]

def test_file_operations():
    # Create a temporary file
    with tempfile.NamedTemporaryFile(mode='w', delete=False) as tmp:
        tmp_name = tmp.name

    try:
        # Test saving data
        test_data = ['apple', 'banana', 'cherry']
        save_data(tmp_name, test_data)

        # Test loading data
        loaded_data = load_data(tmp_name)
        assert loaded_data == test_data

    finally:
        # Clean up
        os.unlink(tmp_name)
```

☞ **Challenge:** Look at this buggy function. Can you write tests that would catch the bugs?

```python
def calculate_discount(price, discount_percent):
    if discount_percent > 100:
        return 0
    return price - (price * discount_percent)
```

Write tests first, then see if you can spot what's wrong. This is called "test-driven development"

20.9. Tips for Better Tests

1. **Test one thing at a time** - each test should focus on a single behavior
2. **Use descriptive test names** - `test_user_login_with_invalid_password` is better than `test_login`
3. **Test the happy path and edge cases** - what happens with empty inputs, None values, etc.
4. **Don't test implementation details** - test what the function does, not how it does it
5. **Keep tests simple** - if your test is complex, maybe your code is too complex
6. **Run tests frequently** - I run them before every commit

Testing might feel like extra work at first, but I promise it will save you hours of debugging later. Start small, add tests as you go, and soon it'll become second nature.

Chapter 21. Author's Note

I hope you found this guide helpful. If you have any feedback, questions, or suggestions, please feel free to reach out to me at: marcus@mkaz.com.

If you did find *Working with Python* helpful, consider supporting it with a small donation. I put a lot of time and effort in putting this all together, pay what you think it is worth. I appreciate it.

- Buy me a coffee
- Donate via PayPal

Chapter 22. References

General

- The Zen of Python
- PEP 8 - Style Guide for Python Code
- Python Documentation

Ch 2: Numbers

- Python Docs – Numeric Types
- Python Docs – Math Module

Ch 3: Strings

- Python Docs – String Methods
- Python Docs – String Module
- Python Tutorial – Strings

Ch 5: Lists

- Python Docs – List Type
- Python Tutorial – More on Lists

Ch 6: Dictionaries

- Python Docs – Dictionary Type
- Python Tutorial – Dictionaries

Ch 7: Control Flow

- Python Tutorial – Control Flow
- Python Docs – Compound Statements

Ch 8: Functions

- Python Tutorial – Defining Functions
- Python Docs – Function Definitions
- PEP 3107 – Function Annotations

Ch 9: Classes

- Python Tutorial – Classes

Ch 10: Python Idioms

- Python Docs – Itertools
- Python Docs – Functools
- Python HOWTO – Functional Programming

Ch 11: Type Hints

- Python Type Hints Documentation
- mypy Documentation
- Pyright Documentation

Ch 12: Collections

- Python Docs – Collections Module

Ch 13: Files

- Python Docs – Built-in open() Function
- Python Docs – Pathlib Module
- Python Docs – OS Path Module

Ch 14: Command-line Arguments

- Python Docs – Argparse Module
- Python Docs – sys.argv
- Python HOWTO – Argparse Tutorial

Ch 15: Dates and Time

- Python Docs – Datetime
- strftime.org – clean date format reference

Ch 16: Regular Expressions

- Python Docs – Regular Expression Operations
- Python HOWTO – Regular Expression
- Python Docs – Regular Expression Syntax

Ch 17: System Commands

- Python Docs – Subprocess Module
- Python Docs – OS Module
- Python Docs – Shell Utilities

Ch 18: Dev Environment

- Python Docs – Virtual Environments
- Pip Documentation
- Python Docs – Installing Python Modules

Ch 19: Project Structure

- Python Packaging Tutorial
- Python Tutorial – Modules

Ch 20: Testing

- Pytest Documentation