Python basics for revision

Math Operators

From **highest** to **lowest** precedence:

Operators	Operation	Example
**	Exponent	`2 ** 3 = 8`
%	Modulus/Remainder	`22 % 8 = 6`
//	Integer division	`22 // 8 = 2`
/	Division	`22 / 8 = 2.75`
*	Multiplication	`3 * 3 = 9`
-	Subtraction	`5 - 2 = 3`
+	Addition	`2 + 2 = 4`

Examples of expressions:

```
>>> 2 + 3 * 6
# 20

>>> (2 + 3) * 6
# 30

>>> 2 ** 8
# 256

>>> 23 // 7
# 3

>>> 23 % 7
# 2
```

```
>>> (5 - 1) * ((7 + 1) / (3 - 1))
# 16.0
```

Augmented Assignment Operators

Operator	Equivalent
`var += 1`	`var = var + 1`
`var -= 1`	`var = var - 1`
`var *= 1`	`var = var * 1`
`var /= 1`	`var = var / 1`
`var //= 1`	`var = var // 1`
`var %= 1`	`var = var % 1`
`var **= 1`	`var = var ** 1`

Examples:

```
>>> greeting = 'Hello'
>>> greeting += ' world!'
>>> greeting
# 'Hello world!'

>>> number = 1
>>> number += 1
>>> number
# 2

>>> my_list = ['item']
>>> my_list *= 3
>>> my_list
# ['item', 'item', 'item']
```

Walrus Operator

The Walrus Operator allows assignment of variables within an expression while returning the value of the variable

Example:

```
>>> print(my_var:="Hello World!")
# 'Hello world!'

>>> my_var="Yes"
>>> print(my_var)
# 'Yes'

>>> print(my_var:="Hello")
# 'Hello'

# Without using the walrus operator
numbers = [5, 8, 2, 10, 3]
n = len(numbers)
if n > 0:
    print(f'The list has {n} elements.')

# With the walrus operator
numbers = [5, 8, 2, 10, 3]
if (n := len(numbers)) > 0:
    print(f'The list has {n} elements.')
```

The *Walrus Operator*, or **Assignment Expression Operator** was firstly introduced in 2018 via PEP 572, and then officially released with **Python 3.8** in October 2019.

Data Types

Data Type	Examples
Integers	`-2, -1, 0, 1, 2, 3, 4, 5`
Floating-point numbers	`-1.25, -1.0, --0.5, 0.0, 0.5, 1.0, 1.25`
Strings	`'a', 'aa', 'aaa', 'Hello!', '11 cats'`

Concatenation and Replication

String concatenation:

```
>>> 'Alice' 'Bob'
# 'AliceBob'
```

String replication:

```
>>> 'Alice' * 5
# 'AliceAliceAliceAliceAlice'
```

Variables

You can name a variable anything as long as it obeys the following rules:

1. It can be only one word.

```
>>> # bad
>>> my variable = 'Hello'

>>> # good
>>> var = 'Hello'
```

2. It can use only letters, numbers, and the underscore (_) character.

```
>>> # bad
>>> %$@variable = 'Hello'

>>> # good
>>> my_var = 'Hello'

>>> # good
>>> my_var_2 = 'Hello'
```

3. It can't begin with a number.

```
>>> # this wont work
>>> 23_var = 'hello'
```

4. Variable name starting with an underscore (_) are considered as "unuseful".

```
>>> # _spam should not be used again in the code
>>> _spam = 'Hello'
```

Comments

Inline comment:

```
# This is a comment
```

Multiline comment:

```
# This is a
# multiline comment
```

Code with a comment:

```
a = 1  # initialization
```

Please note the two spaces in front of the comment.

Function docstring:

```
def foo():
    """
    This is a function docstring
    You can also use:
    ''' Function Docstring '''
    """
```

The print() Function

The `print()` function writes the value of the argument(s) it is given. [...] it handles multiple arguments, floating point-quantities, and strings. Strings are printed without quotes, and a space is inserted between items, so you can format things nicely:

```
>>> print('Hello world!')
# Hello world!

>>> a = 1
>>> print('Hello world!', a)
# Hello world! 1
```

Use a backslash () to continue a statement to the next line

```
>>> total=1+2+3+4+5+6+7+\
4+5+6

>>> print(total)
# 43
```

Multiple Statements on a single line

```
x=5;y=10;z=x+y
print(z)
# 15
```

The end keyword

The keyword argument **end** can be used to avoid the newline after the output, or end the output with a different string:

```
phrase = ['printed', 'with', 'a', 'dash', 'in', 'between']
>>> for word in phrase:
...     print(word, end='-')
...
# printed-with-a-dash-in-between-
```

The sep keyword

The keyword **sep** specify how to separate the objects, if there is more than one:

```
print('cats', 'dogs', 'mice', sep=',')
# cats,dogs,mice
```

The input() Function

This function takes the input from the user and converts it into a string:

```
>>> print('What is your name?')   # ask for their name
>>> my_name = input()
>>> print('Hi, {}'.format(my_name))
# What is your name?
# Martha
# Hi, Martha
```

`input()` can also set a default message without using `print()`:

```
>>> my_name = input('What is your name? ')  # default message
>>> print('Hi, {}'.format(my_name))
# What is your name? Martha
# Hi, Martha
```

It is also possible to use formatted strings to avoid using .format:

```
>>> my_name = input('What is your name? ')  # default message
>>> print(f'Hi, {my_name}')
# What is your name? Martha
# Hi, Martha
```

The len() Function

Evaluates to the integer value of the number of characters in a string, list, dictionary, etc.:

```
>>> len('hello')
# 5

>>> len(['cat', 3, 'dog'])
# 3
```

Test of emptiness example:

```
>>> a = [1, 2, 3]

# bad
>>> if len(a) > 0:  # evaluates to True
...     print("the list is not empty!")
...
# the list is not empty!
# good
>>> if a: # evaluates to True
...     print("the list is not empty!")
...
# the list is not empty!
```

The str(), int(), and float() Functions

These functions allow you to change the type of variable. For example, you can transform from an `integer` or `float` to a `string`:

```
>>> str(29)
# '29'

>>> str(-3.14)
# '-3.14'
```

Or from a `string` to an `integer` or `float`:

```
>>> int('11')
# 11

>>> float('3.14')
# 3.14
```

Python control flow

Control flow is the order in which individual statements, instructions, or function calls are executed or evaluated. The control flow of a Python program is regulated by conditional statements, loops, and function calls.

Comparison Operators

Operator	Meaning
`==`	Equal to
`!=`	Not equal to
`<`	Less than
`>`	Greater Than

Operator	Meaning
`<=`	Less than or Equal to
`>=`	Greater than or Equal to

These operators evaluate to True or False depending on the values you give them.

Examples:

```
>>> 42 == 42
True

>>> 40 == 42
False

>>> 'hello' == 'hello'
True

>>> 'hello' == 'Hello'
False

>>> 'dog' != 'cat'
True

>>> 42 == 42.0
True

>>> 42 == '42'
False
```

Boolean Operators

There are three Boolean operators: `and`, `or`, and `not`. In the order of precedence, highest to lowest they are `not`, `and` and `or`.

The `and` Operator's *Truth* Table:

Expression	Evaluates to
`True and True`	`True`
`True and False`	`False`
`False and True`	`False`
`False and False`	`False`

The `or` Operator's *Truth* Table:

Expression	Evaluates to
`True or True`	`True`
`True or False`	`True`
`False or True`	`True`
`False or False`	`False`

The `not` Operator's *Truth* Table:

Expression	Evaluates to
`not True`	`False`
`not False`	`True`

Mixing Operators

You can mix boolean and comparison operators:

```
>>> (4 < 5) and (5 < 6)
True

>>> (4 < 5) and (9 < 6)
False

>>> (1 == 2) or (2 == 2)
True
```

Also, you can mix use multiple Boolean operators in an expression, along with the comparison operators:

```
>>> 2 + 2 == 4 and not 2 + 2 == 5 and 2 * 2 == 2 + 2
True
>>> 5 > 4 or 3 < 4 and 5 > 5
True
>>> (5 > 4 or 3 < 4) and 5 > 5
False
```

Explanation: - The first expression checks multiple conditions: - `2 + 2 == 4` evaluates to `True`. - `not 2 + 2 == 5` evaluates to `True` since `2 + 2 == 5` is `False`. - `2 * 2 == 2 + 2` evaluates to `True`. - Combining all with `and` results in `True`.

- In the second expression:
    - The sub-expression `3 < 4 and 5 > 5` evaluates to `False` (`True and False`).
    - `5 > 4` evaluates to `True`.
    - With `True or False`, the overall result is `True`.
- For the third expression:
    - The parenthetical expression `(5 > 4 or 3 < 4)` evaluates to `True` (`True or False`).
    - `5 > 5` evaluates to `False`.
    - Combining with `and`, `True and False` results in `False`.

if Statements

The `if` statement evaluates an expression, and if that expression is `True`, it then executes the following indented code:

```
>>> name = 'Debora'

>>> if name == 'Debora':
...     print('Hi, Debora')
...
# Hi, Debora

>>> if name != 'George':
...     print('You are not George')
...
# You are not George
```

The `else` statement executes only if the evaluation of the `if` and all the `elif` expressions are `False`:

```
>>> name = 'Debora'

>>> if name == 'George':
...     print('Hi, George.')
... else:
...     print('You are not George')
...
# You are not George
```

Only after the if statement expression is False, the elif statement is evaluated and executed:

```
>>> name = 'George'

>>> if name == 'Debora':
...     print('Hi Debora!')
... elif name == 'George':
...     print('Hi George!')
...
# Hi George!
```

the elif and else parts are optional.

```
>>> name = 'Antony'

>>> if name == 'Debora':
...     print('Hi Debora!')
... elif name == 'George':
...     print('Hi George!')
... else:
...     print('Who are you?')
...
# Who are you?
```

Ternary Conditional Operator

Many programming languages have a ternary operator, which define a conditional expression. The most common usage is to make a terse, simple conditional assignment statement. In other words, it offers one-line code to evaluate the first expression if the condition is true, and otherwise it evaluates the second expression.

```
<expression1> if <condition> else <expression2>
```

Example:

```
>>> age = 15

>>> # this if statement:
>>> if age < 18:
```

```
...     print('kid')
... else:
...     print('adult')
...
# output: kid

>>> # is equivalent to this ternary operator:
>>> print('kid' if age < 18 else 'adult')
# output: kid
```

Ternary operators can be chained:

```
>>> age = 15

>>> # this ternary operator:
>>> print('kid' if age < 13 else 'teen' if age < 18 else 'adult')

>>> # is equivalent to this if statement:
>>> if age < 18:
...     if age < 13:
...         print('kid')
...     else:
...         print('teen')
... else:
...     print('adult')
...
# output: teen
```

Switch-Case Statement

In computer programming languages, a switch statement is a type of selection control mechanism used to allow the value of a variable or expression to change the control flow of program execution via search and map.

The *Switch-Case statements*, or **Structural Pattern Matching**, was firstly introduced in 2020 via PEP 622, and then officially released with **Python 3.10** in September 2022. The PEP 636 provides an official tutorial for the Python Pattern matching or Switch-Case statements.

Matching single values

```
>>> response_code = 201
>>> match response_code:
...     case 200:
...         print("OK")
...     case 201:
...         print("Created")
...     case 300:
...         print("Multiple Choices")
...     case 307:
...         print("Temporary Redirect")
...     case 404:
...         print("404 Not Found")
...     case 500:
...         print("Internal Server Error")
...     case 502:
...         print("502 Bad Gateway")
...
# Created
```

Matching with the or Pattern

In this example, the pipe character (`|` or `or`) allows python to return the same response for two or more cases.

```
>>> response_code = 502
>>> match response_code:
...     case 200 | 201:
...         print("OK")
...     case 300 | 307:
...         print("Redirect")
...     case 400 | 401:
...         print("Bad Request")
...     case 500 | 502:
...         print("Internal Server Error")
...
# Internal Server Error
```

Matching by the length of an Iterable

```
>>> today_responses = [200, 300, 404, 500]
>>> match today_responses:
...     case [a]:
...             print(f"One response today: {a}")
...     case [a, b]:
...             print(f"Two responses today: {a} and {b}")
...     case [a, b, *rest]:
...             print(f"All responses: {a}, {b}, {rest}")
...
# All responses: 200, 300, [404, 500]
```

Default value

The underscore symbol (_) is used to define a default case:

```
>>> response_code = 800
>>> match response_code:
...     case 200 | 201:
...         print("OK")
...     case 300 | 307:
...         print("Redirect")
...     case 400 | 401:
...         print("Bad Request")
...     case 500 | 502:
...         print("Internal Server Error")
...     case _:
...         print("Invalid Code")
...
# Invalid Code
```

Matching Builtin Classes

```
>>> response_code = "300"
>>> match response_code:
...     case int():
...             print('Code is a number')
...     case str():
```

```
...             print('Code is a string')
...     case _:
...             print('Code is neither a string nor a number')
...
# Code is a string
```

Guarding Match-Case Statements

```
>>> response_code = 300
>>> match response_code:
...     case int():
...             if response_code > 99 and response_code < 500:
...                 print('Code is a valid number')
...     case _:
...             print('Code is an invalid number')
...
# Code is a valid number
```

while Loop Statements

The while statement is used for repeated execution as long as an expression is `True`:

```
>>> spam = 0
>>> while spam < 5:
...     print('Hello, world.')
...     spam = spam + 1
...
# Hello, world.
# Hello, world.
# Hello, world.
# Hello, world.
# Hello, world.
```

break Statements

If the execution reaches a `break` statement, it immediately exits the `while` loop's clause:

```
>>> while True:
...     name = input('Please type your name: ')
...     if name == 'your name':
...         break
...
>>> print('Thank you!')
# Please type your name: your name
# Thank you!
```

continue Statements

When the program execution reaches a `continue` statement, the program execution immediately jumps back to the start of the loop(Skip the current exection).

```
>>> while True:
...     name = input('Who are you? ')
...     if name != 'Joe':
...         continue
...     password = input('Password? (It is a fish.): ')
...     if password == 'swordfish':
...         break
...
>>> print('Access granted.')
# Who are you? Charles
# Who are you? Debora
# Who are you? Joe
# Password? (It is a fish.): swordfish
# Access granted.
```

The pass Statement

The `pass` statement in Python is a null operation; it doesn't do anything but is syntactically required. It's useful as a placeholder in code where syntax requires a statement but no action is necessary. For example, it can be used in function or class definitions where you want to implement the body later.

```
def my_function():
    pass  # This function does nothing for now

class MyClass:
```

~~~
    pass  # This class has no attributes or methods yet

## For loop

The `for` loop iterates over a `list`, `tuple`, `dictionary`,
`set` or `string`:

```python
>>> pets = ['Bella', 'Milo', 'Loki']
>>> for pet in pets:
...     print(pet)
...
# Bella
# Milo
# Loki
~~~

The range() function

The `range()` function returns a sequence of numbers. It starts from 0, increments by 1, and stops before a specified number:

```
>>> for i in range(5):
...     print(f'Will stop at 5! or 4? ({i})')
...
# Will stop at 5! or 4? (0)
# Will stop at 5! or 4? (1)
# Will stop at 5! or 4? (2)
# Will stop at 5! or 4? (3)
# Will stop at 5! or 4? (4)
```

The `range()` function can also modify its 3 defaults arguments. The first two will be the `start` and `stop` values, and the third will be the `step` argument. The step is the amount that the variable is increased by after each iteration.

```
# range(start, stop, step)
>>> for i in range(0, 10, 2):
...    print(i)
...
# 0
# 2
# 4
```

```
# 6
# 8
```

You can even use a negative number for the step argument to make the for loop count down instead of up.

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

For else statement

This allows to specify a statement to execute in case of the full loop has been executed. Only useful when a `break` condition can occur in the loop:

```
>>> for i in [1, 2, 3, 4, 5]:
...    if i == 3:
...        break
... else:
...    print("only executed when no item is equal to 3")
```

Ending a Program with sys.exit()

`exit()` function allows exiting Python.

```
>>> import sys

>>> while True:
...     feedback = input('Type exit to exit: ')
...     if feedback == 'exit':
...         print(f'You typed {feedback}.')
...         sys.exit()
...
# Type exit to exit: open
# Type exit to exit: close
# Type exit to exit: exit
# You typed exit
```

```
## Determine if a number is even ,odd, negative

# num=int(input("Enter the number"))
num = 3

if num>0:
    print("The number is positive")
    if num%2==0:
        print("The number is even")
    else:
        print("The number is odd")

else:
    print("The number is zero or negative")
```

```
The number is positive
The number is odd
```

```
# Determine if a year is a leap year
# year = int(input("Enter the year: "))
year = 2024

# Determine if the year is a leap year
if (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0):
    print(year, "is a leap year")
else:
    print(year, "is not a leap year")
```

```
2024 is a leap year
```

```
## Loop Control Statements

## break
## The break statement exits the loop permaturely

## break sstatement
```

```python
for i in range(10):
    if i==5:
        break
    print(i)
```

```
0
1
2
3
4
```

```python
## continue

## The continue statement skips the current iteration and
# continues with the next.

for i in range(10):
    if i%2==0:
        continue
    print(i)
```

```
1
3
5
7
9
```

```python
## pass
## The pass statement is a null operation; it does nothing.

for i in range(5):
    if i==3:
        pass
    print(i)
```

```
0
1
2
3
4
```

```
## Nested loopss
## a loop inside a loop

for i in range(3):
    for j in range(2):
        print(f"i:{i} and j:{j}")
```

```
i:0 and j:0
i:0 and j:1
i:1 and j:0
i:1 and j:1
i:2 and j:0
i:2 and j:1
```

```
## Examples- Calculate the sum of first N natural numbers using
a while and for loop

## while loop

n=10
sum_=0
count=1

while count<=n:
    sum_=sum_+count
    count=count+1

print("Sum of first 10 natural number:",sum_)
```

```
SyntaxError: invalid syntax (1958771481.py, line 2)
```

```
# For loop
n=10
sum_1=0
for i in range(n+1):
    sum_1=sum_1+i

print(sum_1)
```

Python Lists

Lists are one of the 4 data types(List, tuples, set, dictionary) in Python used to store collections of data.

```
['John', 'Peter', 'Debora', 'Charles']
```

Getting values with indexes

```
>>> furniture = ['table', 'chair', 'rack', 'shelf']

>>> furniture[0]
# 'table'

>>> furniture[1]
# 'chair'

>>> furniture[2]
# 'rack'

>>> furniture[3]
# 'shelf'
```

Negative indexes

```
>>> furniture = ['table', 'chair', 'rack', 'shelf']

>>> furniture[-1]
# 'shelf'

>>> furniture[-3]
# 'chair'

>>> f'The {furniture[-1]} is bigger than the {furniture[-3]}'
# 'The shelf is bigger than the chair'
```

Getting sublists with Slices

```
>>> furniture = ['table', 'chair', 'rack', 'shelf']

>>> furniture[0:4]
# ['table', 'chair', 'rack', 'shelf']

>>> furniture[1:3]
# ['chair', 'rack']

>>> furniture[0:-1]
# ['table', 'chair', 'rack']

>>> furniture[:2]
# ['table', 'chair']

>>> furniture[1:]
# ['chair', 'rack', 'shelf']

>>> furniture[:]
# ['table', 'chair', 'rack', 'shelf']
```

Slicing the complete list will perform a copy:

```
>>> spam2 = spam[:]
# ['cat', 'bat', 'rat', 'elephant']

>>> spam.append('dog')
>>> spam
# ['cat', 'bat', 'rat', 'elephant', 'dog']

>>> spam2
# ['cat', 'bat', 'rat', 'elephant']
```

Getting a list length with len()

```
>>> furniture = ['table', 'chair', 'rack', 'shelf']
>>> len(furniture)
# 4
```

Changing values with indexes

```
>>> furniture = ['table', 'chair', 'rack', 'shelf']

>>> furniture[0] = 'desk'
>>> furniture
# ['desk', 'chair', 'rack', 'shelf']

>>> furniture[2] = furniture[1]
>>> furniture
# ['desk', 'chair', 'chair', 'shelf']

>>> furniture[-1] = 'bed'
>>> furniture
# ['desk', 'chair', 'chair', 'bed']
```

Concatenation and Replication

```
>>> [1, 2, 3] + ['A', 'B', 'C']
# [1, 2, 3, 'A', 'B', 'C']

>>> ['X', 'Y', 'Z'] * 3
# ['X', 'Y', 'Z', 'X', 'Y', 'Z', 'X', 'Y', 'Z']

>>> my_list = [1, 2, 3]
>>> my_list = my_list + ['A', 'B', 'C']
>>> my_list
# [1, 2, 3, 'A', 'B', 'C']
```

Using for loops with Lists

```
>>> furniture = ['table', 'chair', 'rack', 'shelf']

>>> for item in furniture:
...     print(item)
# table
# chair
# rack
# shelf
```

Getting the index in a loop with enumerate()

```
>>> furniture = ['table', 'chair', 'rack', 'shelf']

>>> for index, item in enumerate(furniture):
...     print(f'index: {index} - item: {item}')
# index: 0 - item: table
# index: 1 - item: chair
# index: 2 - item: rack
# index: 3 - item: shelf
```

Loop in Multiple Lists with zip()

```
>>> furniture = ['table', 'chair', 'rack', 'shelf']
>>> price = [100, 50, 80, 40]

>>> for item, amount in zip(furniture, price):
...     print(f'The {item} costs ${amount}')
# The table costs $100
# The chair costs $50
# The rack costs $80
# The shelf costs $40
```

The in and not in operators

```
>>> 'rack' in ['table', 'chair', 'rack', 'shelf']
# True

>>> 'bed' in ['table', 'chair', 'rack', 'shelf']
# False

>>> 'bed' not in furniture
# True
```

```
>>> 'rack' not in furniture
# False
```

The Multiple Assignment Trick

The multiple assignment trick is a shortcut that lets you assign multiple variables with the values in a list in one line of code. So instead of doing this:

```
>>> furniture = ['table', 'chair', 'rack', 'shelf']
>>> table = furniture[0]
>>> chair = furniture[1]
>>> rack = furniture[2]
>>> shelf = furniture[3]
```

You could type this line of code:

```
>>> furniture = ['table', 'chair', 'rack', 'shelf']
>>> table, chair, rack, shelf = furniture

>>> table
# 'table'

>>> chair
# 'chair'

>>> rack
# 'rack'

>>> shelf
# 'shelf'
```

The multiple assignment trick can also be used to swap the values in two variables:

```
>>> a, b = 'table', 'chair'
>>> a, b = b, a
>>> print(a)
# chair

>>> print(b)
# table
```

The index Method

The `index` method allows you to find the index of a value by passing its name:

```
>>> furniture = ['table', 'chair', 'rack', 'shelf']
>>> furniture.index('chair')
# 1
```

Adding Values

append()

`append` adds an element to the end of a `list`:

```
>>> furniture = ['table', 'chair', 'rack', 'shelf']
>>> furniture.append('bed')
>>> furniture
# ['table', 'chair', 'rack', 'shelf', 'bed']
```

insert()

`insert` adds an element to a `list` at a given position:

```
>>> furniture = ['table', 'chair', 'rack', 'shelf']
>>> furniture.insert(1, 'bed')
>>> furniture
# ['table', 'bed', 'chair', 'rack', 'shelf']
```

Removing Values

del()

`del` removes an item using the index:

```
>>> furniture = ['table', 'chair', 'rack', 'shelf']
>>> del furniture[2]
>>> furniture
# ['table', 'chair', 'shelf']

>>> del furniture[2]
>>> furniture
# ['table', 'chair']
```

remove()

`remove` removes an item with using actual value of it:

```
>>> furniture = ['table', 'chair', 'rack', 'shelf']
>>> furniture.remove('chair')
>>> furniture
# ['table', 'rack', 'shelf']
```

pop()

By default, `pop` will remove and return the last item of the list. You can also pass the index of the element as an optional parameter:

```
>>> animals = ['cat', 'bat', 'rat', 'elephant']

>>> animals.pop()
'elephant'

>>> animals
['cat', 'bat', 'rat']

>>> animals.pop(0)
'cat'

>>> animals
['bat', 'rat']
```

Sorting values with sort()

```
>>> numbers = [2, 5, 3.14, 1, -7]
>>> numbers.sort()
>>> numbers
# [-7, 1, 2, 3.14, 5]
```

```
furniture = ['table', 'chair', 'rack', 'shelf']
furniture.sort()
furniture
# ['chair', 'rack', 'shelf', 'table']
```

You can also pass `True` for the `reverse` keyword argument to have `sort()` sort the values in reverse order:

```
>>> furniture.sort(reverse=True)
>>> furniture
# ['table', 'shelf', 'rack', 'chair']
```

If you need to sort the values in regular alphabetical order, pass `str.lower` for the key keyword argument in the sort() method call:

```
>>> letters = ['a', 'z', 'A', 'Z']
>>> letters.sort(key=str.lower)
>>> letters
# ['a', 'A', 'z', 'Z']
```

You can use the built-in function `sorted` to return a new list:

```
>>> furniture = ['table', 'chair', 'rack', 'shelf']
>>> sorted(furniture)
# ['chair', 'rack', 'shelf', 'table']
```

```
## Slicing List
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(numbers[2:5])
print(numbers[:5])
print(numbers[5:])
print(numbers[::2])
print(numbers[::-1])
```

The Tuple data type

Tuples vs Lists The key difference between tuples and lists is that, while tuples are immutable objects, lists are mutable. This means that tuples cannot be changed while the lists can be modified. Tuples are more memory efficient than the lists.

```
>>> furniture = ('table', 'chair', 'rack', 'shelf')

>>> furniture[0]
# 'table'

>>> furniture[1:3]
# ('chair', 'rack')

>>> len(furniture)
# 4
```

The main way that tuples are different from lists is that tuples, like strings, are immutable.

Converting between list() and tuple()

```
>>> tuple(['cat', 'dog', 5])
# ('cat', 'dog', 5)

>>> list(('cat', 'dog', 5))
# ['cat', 'dog', 5]

>>> list('hello')
# ['h', 'e', 'l', 'l', 'o']
```

```
## creating a tuple
empty_tuple=()
print(empty_tuple)
print(type(empty_tuple))
```

```
lst=list()
print(type(lst))
tpl=tuple()
print(type(tpl))
```

```
numbers=tuple([1,2,3,4,5,6])
numbers
```

```
list((1,2,3,4,5,6))
```

```
mixed_tuple=(1,"Hello World",3.14, True)
print(mixed_tuple)
```

```
print(numbers[2])
print(numbers[-1])
```

```
numbers[0:4]
```

```
numbers[::-1]
```

```
## Tuple Operations
concatenation_tuple=numbers + mixed_tuple
print(concatenation_tuple)
```

```
mixed_tuple * 3
```

```
numbers *3
```

```
## Immutable Nature Of Tuples
## Tuples are immutable, meaning their elements cannot be changed once assigned.

lst=[1,2,3,4,5]
print(lst)

lst[1]="Mukesh"
print(lst)
```

```
# numbers[1]="Mukesh"
# TypeError: 'tuple' object does not support item assignment
```

```
numbers
```

```
## Tuple Methods
print(numbers.count(1))
print(numbers.index(3))
```

```
## Packing and Unpacking tuple
## packing
packed_tuple=1,"Hello",3.14
print(packed_tuple)
```

```
##unpacking a tuple
a,b,c=packed_tuple

print(a)
print(b)
print(c)
```

```
## Unpacking with *
numbers=(1,2,3,4,5,6)
first,*middle,last=numbers
print(first)
print(middle)
print(last)
```

```
## Nested Tuple
## Nested List
lst=[[1,2,3,4],[6,7,8,9],[1,"Hello",3.14,"c"]]
lst[0][0:3]
```

```
lst=[[1,2,3,4],[6,7,8,9],(1,"Hello",3.14,"c")]
lst[2][0:3]
```

```
nested_tuple = ((1, 2, 3), ("a", "b", "c"), (True, False))

## access the elements inside a tuple
print(nested_tuple[0])
print(nested_tuple[1][2])
```

```
## iterating over nested tuples
for sub_tuple in nested_tuple:
    for item in sub_tuple:
        print(item,end=" ,")
    print()
```

Python Sets

Python comes equipped with several built-in data types to help us organize our data. These structures include lists, dictionaries, tuples and **sets**.

From the Python 3 documentation A set is an unordered collection with no duplicate elements. Basic uses include membership testing and eliminating duplicate entries.

Initializing a set

There are two ways to create sets: using curly braces `{}` and the built-in function `set()`

Empty Sets When creating set, be sure to not use empty curly braces {} or you will get an empty dictionary instead.

```
>>> s = {1, 2, 3}
>>> s = set([1, 2, 3])

>>> s = {}  # this will create a dictionary instead of a set
>>> type(s)
# <class 'dict'>
```

Unordered collections of unique elements

A set automatically remove all the duplicate values.

```
>>> s = {1, 2, 3, 2, 3, 4}
>>> s
# {1, 2, 3, 4}
```

And as an unordered data type, they can't be indexed.

```
>>> s = {1, 2, 3}
>>> s[0]
# Traceback (most recent call last):
#   File "<stdin>", line 1, in <module>
# TypeError: 'set' object does not support indexing
```

set add and update

Using the `add()` method we can add a single element to the set.

```
>>> s = {1, 2, 3}
>>> s.add(4)
>>> s
# {1, 2, 3, 4}
```

And with `update()`, multiple ones:

```
>>> s = {1, 2, 3}
>>> s.update([2, 3, 4, 5, 6])
>>> s
# {1, 2, 3, 4, 5, 6}
```

set remove and discard

Both methods will remove an element from the set, but `remove()` will raise a `key error` if the value doesn't exist.

```
>>> s = {1, 2, 3}
>>> s.remove(3)
>>> s
# {1, 2}

>>> s.remove(3)
# Traceback (most recent call last):
#   File "<stdin>", line 1, in <module>
# KeyError: 3
```

`discard()` won't raise any errors.

```
>>> s = {1, 2, 3}
>>> s.discard(3)
>>> s
# {1, 2}
>>> s.discard(3)
```

set union

`union()` or `|` will create a new set with all the elements from the sets provided.

```
>>> s1 = {1, 2, 3}
>>> s2 = {3, 4, 5}
>>> s1.union(s2)  # or 's1 | s2'
# {1, 2, 3, 4, 5}
```

set intersection

`intersection()` or `&` will return a set with only the elements that are common to all of them.

```
>>> s1 = {1, 2, 3}
>>> s2 = {2, 3, 4}
>>> s3 = {3, 4, 5}
>>> s1.intersection(s2, s3)  # or 's1 & s2 & s3'
# {3}
```

set difference

`difference()` or `-` will return only the elements that are unique to the first set (invoked set).

```
>>> s1 = {1, 2, 3}
>>> s2 = {2, 3, 4}

>>> s1.difference(s2)  # or 's1 - s2'
# {1}

>>> s2.difference(s1) # or 's2 - s1'
# {4}
```

set symmetric_difference

`symmetric_difference()` or `^` will return all the elements that are not common between them.

```
>>> s1 = {1, 2, 3}
>>> s2 = {2, 3, 4}
>>> s1.symmetric_difference(s2)  # or 's1 ^ s2'
# {1, 4}
```

Python Dictionaries

In Python, a dictionary is an *ordered* (from Python > 3.7) collection of `key: value` pairs.

From the Python 3 documentation The main operations on a dictionary are storing a value with some key and extracting the value given the key. It is also possible to delete a key:value pair with del.

Example Dictionary:

```
my_cat = {
    'size': 'fat',
    'color': 'gray',
    'disposition': 'loud'
}
```

Set key, value using subscript operator []

```
>>> my_cat = {
...  'size': 'fat',
...  'color': 'gray',
...  'disposition': 'loud',
... }
>>> my_cat['age_years'] = 2
>>> print(my_cat)
...
# {'size': 'fat', 'color': 'gray', 'disposition': 'loud', 'age_years': 2}
```

Get value using subscript operator []

In case the key is not present in dictionary `KeyError` is raised.

```
>>> my_cat = {
...  'size': 'fat',
...  'color': 'gray',
...  'disposition': 'loud',
... }
>>> print(my_cat['size'])
...
# fat
>>> print(my_cat['eye_color'])
# Traceback (most recent call last):
#   File "<stdin>", line 1, in <module>
# KeyError: 'eye_color'
```

values()

The `values()` method gets the **values** of the dictionary:

```
>>> pet = {'color': 'red', 'age': 42}
>>> for value in pet.values():
...     print(value)
...
# red
# 42
```

keys()

The `keys()` method gets the **keys** of the dictionary:

```
>>> pet = {'color': 'red', 'age': 42}
>>> for key in pet.keys():
...     print(key)
...
# color
# age
```

There is no need to use **.keys()** since by default you will loop through keys:

```
>>> pet = {'color': 'red', 'age': 42}
>>> for key in pet:
...     print(key)
...
# color
# age
```

items()

The `items()` method gets the **items** of a dictionary and returns them as a Tuple:

```
>>> pet = {'color': 'red', 'age': 42}
>>> for item in pet.items():
...     print(item)
...
# ('color', 'red')
# ('age', 42)
```

Using the `keys()`, `values()`, and `items()` methods, a for loop can iterate over the keys, values, or key-value pairs in a dictionary, respectively.

```
>>> pet = {'color': 'red', 'age': 42}
>>> for key, value in pet.items():
...     print(f'Key: {key} Value: {value}')
...
# Key: color Value: red
# Key: age Value: 42
```

get()

The `get()` method returns the value of an item with the given key. If the key doesn't exist, it returns `None`:

```
>>> wife = {'name': 'Rose', 'age': 33}

>>> f'My wife name is {wife.get("name")}'
# 'My wife name is Rose'

>>> f'She is {wife.get("age")} years old.'
# 'She is 33 years old.'

>>> f'She is deeply in love with {wife.get("husband")}'
# 'She is deeply in love with None'
```

You can also change the default `None` value to one of your choice:

```
>>> wife = {'name': 'Rose', 'age': 33}

>>> f'She is deeply in love with {wife.get("husband", "lover")}'
# 'She is deeply in love with lover'
```

Adding items with setdefault()

It's possible to add an item to a dictionary in this way:

```
>>> wife = {'name': 'Rose', 'age': 33}
>>> if 'has_hair' not in wife:
...     wife['has_hair'] = True
```

Using the `setdefault` method, we can make the same code more short:

```
>>> wife = {'name': 'Rose', 'age': 33}
>>> wife.setdefault('has_hair', True)
>>> wife
# {'name': 'Rose', 'age': 33, 'has_hair': True}
```

Removing Items

pop()

The `pop()` method removes and returns an item based on a given key.

```
>>> wife = {'name': 'Rose', 'age': 33, 'hair': 'brown'}
>>> wife.pop('age')
# 33
>>> wife
# {'name': 'Rose', 'hair': 'brown'}
```

popitem()

The `popitem()` method removes the last item in a dictionary and returns it.

```
>>> wife = {'name': 'Rose', 'age': 33, 'hair': 'brown'}
>>> wife.popitem()
# ('hair', 'brown')
>>> wife
# {'name': 'Rose', 'age': 33}
```

del()

The `del()` method removes an item based on a given key.

```
>>> wife = {'name': 'Rose', 'age': 33, 'hair': 'brown'}
>>> del wife['age']
>>> wife
# {'name': 'Rose', 'hair': 'brown'}
```

clear()

The`clear()` method removes all the items in a dictionary.

```
>>> wife = {'name': 'Rose', 'age': 33, 'hair': 'brown'}
>>> wife.clear()
>>> wife
# {}
```

Checking keys in a Dictionary

```
>>> person = {'name': 'Rose', 'age': 33}

>>> 'name' in person.keys()
# True

>>> 'height' in person.keys()
# False

>>> 'skin' in person # You can omit keys()
# False
```

Checking values in a Dictionary

```
>>>  person = {'name': 'Rose', 'age': 33}

>>> 'Rose' in person.values()
# True

>>> 33 in person.values()
# True
```

Pretty Printing

```
>>> import pprint

>>> wife = {'name': 'Rose', 'age': 33, 'has_hair': True,
            'hair_color': 'brown', 'height': 1.6,
            'eye_color': 'brown'}
>>> pprint.pprint(wife)
# {'age': 33,
#  'eye_color': 'brown',
#  'hair_color': 'brown',
#  'has_hair': True,
#  'height': 1.6,
#  'name': 'Rose'}
```

Merge two dictionaries

For Python 3.5+:

```
>>> dict_a = {'a': 1, 'b': 2}
>>> dict_b = {'b': 3, 'c': 4}
>>> dict_c = {**dict_a, **dict_b}
>>> dict_c
# {'a': 1, 'b': 3, 'c': 4}
```

Merging Process: The unpacking operator ** is used to unpack the dictionary items into a new dictionary. When using {**dict1,** dict2}, each key-value pair from dict1 is added first, followed by each key-value pair from dict2.

Key Overlap: If dict1 and dict2 have overlapping keys, the values from dict2 will overwrite those from dict1 for those keys. In this case, both dict1 and dict2 contain the key "b", but the value in merged_dict is 3, taken from dict2, because dict2 is unpacked after dict1

```
## Creating Dictionaries
empty_dict={}
print(type(empty_dict))
```

```
empty_dict=dict()
empty_dict
```

```
## Dictionary methods
student={"name":"Mukesh","age":27,"grade":'A'}
keys=student.keys() ##get all the keys
print(keys)
values=student.values() ##get all values
print(values)
```

```
items=student.items() ##get all key value pairs
print(items)
```

shallow copy vs deep copy The main difference between a shallow copy and a deep copy is that a shallow copy references the original data, while a deep copy creates an independent copy of the original object:

```
## Dictionary Comphrehension
squares={x:x**2 for x in range(5)}
print(squares)
```

```
## Condition dictionary comprehension
evens={x:x**2 for x in range(10) if x%2==0}
print(evens)
```

```
## Practical Examples

## USe a dictionary to count he frequency of elements in list

numbers=[1,2,2,3,3,3,4,4,4,4]
frequency={}

for number in numbers:
    if number in frequency:
        frequency[number]+=1
    else:
        frequency[number]=1
print(frequency)
```

Python Comprehensions

List Comprehensions are a special kind of syntax that let us create lists out of other lists, and are incredibly useful when dealing with numbers and with one or two levels of nested for loops.

From the Python 3 tutorial List comprehensions provide a concise way to create lists. [...] or to create a subsequence of those elements that satisfy a certain condition.

List comprehension

This is how we create a new list from an existing collection with a For Loop:

```
>>> names = ['Charles', 'Susan', 'Patrick', 'George']

>>> new_list = []
>>> for n in names:
...     new_list.append(n)
...
>>> new_list
# ['Charles', 'Susan', 'Patrick', 'George']
```

And this is how we do the same with a List Comprehension:

```
>>> names = ['Charles', 'Susan', 'Patrick', 'George']

>>> new_list = [n for n in names]
>>> new_list
# ['Charles', 'Susan', 'Patrick', 'George']
```

We can do the same with numbers:

```
>>> n = [(a, b) for a in range(1, 3) for b in range(1, 3)]
>>> n
# [(1, 1), (1, 2), (2, 1), (2, 2)]
```

Adding conditionals

If we want `new_list` to have only the names that start with C, with a for loop, we would do it like this:

```
>>> names = ['Charles', 'Susan', 'Patrick', 'George', 'Carol']

>>> new_list = []
>>> for n in names:
...     if n.startswith('C'):
...         new_list.append(n)
...
>>> print(new_list)
# ['Charles', 'Carol']
```

In a List Comprehension, we add the `if` statement at the end:

```
>>> new_list = [n for n in names if n.startswith('C')]
>>> print(new_list)
# ['Charles', 'Carol']
```

To use an `if-else` statement in a List Comprehension:

```
>>> nums = [1, 2, 3, 4, 5, 6]
>>> new_list = [num*2 if num % 2 == 0 else num for num in nums]
>>> print(new_list)
# [1, 4, 3, 8, 5, 12]
```

Set and Dict comprehensions The basics of `list` comprehensions also apply to sets and dictionaries.

Set comprehension

```
>>> b = {"abc", "def"}
>>> {s.upper() for s in b}
{"ABC", "DEF"}
```

Dict comprehension

```
>>> c = {'name': 'Pooka', 'age': 5}
>>> {v: k for k, v in c.items()}
{'Pooka': 'name', 5: 'age'}
```

A List comprehension can be generated from a dictionary:

```
>>> c = {'name': 'Pooka', 'age': 5}
>>> ["{}:{}".format(k.upper(), v) for k, v in c.items()]
['NAME:Pooka', 'AGE:5']
```

Python Functions

A function is a block of organized code that is used to perform a single task. They provide better modularity for your application and reuse-ability.

Function Arguments

A function can take `arguments` and `return values`:

In the following example, the function **say__hello** receives the argument "name" and prints a greeting:

```
>>> def say_hello(name):
...     print(f'Hello {name}')
...
>>> say_hello('Carlos')
# Hello Carlos

>>> say_hello('Wanda')
# Hello Wanda

>>> say_hello('Rose')
# Hello Rose
```

Keyword Arguments

To improve code readability, we should be as explicit as possible. We can achieve this in our functions by using `Keyword Arguments`:

```
>>> def say_hi(name, greeting):
...     print(f"{greeting} {name}")
...
>>> # without keyword arguments
>>> say_hi('John', 'Hello')
# Hello John

>>> # with keyword arguments
>>> say_hi(name='Anna', greeting='Hi')
# Hi Anna
```

Return Values

When creating a function using the `def` statement, you can specify what the return value should be with a `return` statement. A return statement consists of the following:

- The `return` keyword.

- The value or expression that the function should return.

```
>>> def sum_two_numbers(number_1, number_2):
...    return number_1 + number_2
...
>>> result = sum_two_numbers(7, 8)
>>> print(result)
# 15
```

Local and Global Scope

- Code in the global scope cannot use any local variables.
- However, a local scope can access global variables.
- Code in a function's local scope cannot use variables in any other local scope.
- You can use the same name for different variables if they are in different scopes. That is, there can be a local variable named spam and a global variable also named spam.

```
global_variable = 'I am available everywhere'

>>> def some_function():
...     print(global_variable)  # because is global
...     local_variable = "only available within this function"
...     print(local_variable)
...
>>> # the following code will throw error because
>>> # 'local_variable' only exists inside 'some_function'
>>> print(local_variable)
Traceback (most recent call last):
  File "<stdin>", line 10, in <module>
NameError: name 'local_variable' is not defined
```

The global Statement

If you need to modify a global variable from within a function, use the global statement:

```
>>> def spam():
...     global eggs
...     eggs = 'spam'
...
>>> eggs = 'global'
>>> spam()
>>> print(eggs)
```

There are four rules to tell whether a variable is in a local scope or global scope:

1. If a variable is being used in the global scope (that is, outside all functions), then it is always a global variable.
2. If there is a global statement for that variable in a function, it is a global variable.
3. Otherwise, if the variable is used in an assignment statement in the function, it is a local variable.
4. But if the variable is not used in an assignment statement, it is a global variable.

Lambda Functions

In Python, a lambda function is a single-line, anonymous function, which can have any number of arguments, but it can only have one expression.

lambda is a minimal function definition that can be used inside an expression. Unlike FunctionDef, body holds a single node.

Single line expression Lambda functions can only evaluate an expression, like a single line of code.

This function:

```
>>> def add(x, y):
...     return x + y
...
>>> add(5, 3)
# 8
```

Is equivalent to the *lambda* function:

```
>>> add = lambda x, y: x + y
>>> add(5, 3)
# 8
```

Like regular nested functions, lambdas also work as lexical closures:

```
>>> def make_adder(n):
...     return lambda x: x + n
...
>>> plus_3 = make_adder(3)
>>> plus_5 = make_adder(5)

>>> plus_3(4)
# 7
>>> plus_5(4)
# 9
```

```
## syntax
def function_name(parameters):
    """Docstring"""
    # Function body
    return expression
```

```
### Variable Length Arguments
## Positional And Keywords arguments

def print_numbers(*Mukesh):
    for number in Mukesh:
        print(number)
```

```
print_numbers(1,2,3,4,5,6,7,8,"Mukesh")
```

```
### Keywords Arguments

def print_details(**kwargs):
    for key,value in kwargs.items():
        print(f"{key}:{value}")
```

```
print_details(name="Mukesh",age="27",country="India")
```

```
def print_details(*args,**kwargs):
    for val in args:
        print(f" Positional arument :{val}")

    for key,value in kwargs.items():
        print(f"{key}:{value}")
```

```python
print_details(1,2,3,4,"Mukesh",name="Mukesh",age="26",country="India")
```

```python
### Return multiple parameters
def multiply(a,b):
    return a*b,a

multiply(2,3)
```

```python
#Syntax
lambda arguments: expression
```

```python
even1=lambda num:num%2==0
even1(12)
```

```python
addition1=lambda x,y,z:x+y+z
addition1(12,13,14)
```

```python
numbers=[1,2,3,4,5,6]
list(map(lambda x:x**2,numbers))
```

The map() Function in Python

The map() function applies a given function to all items in an input list (or any other iterable) and returns a map object (an iterator). This is particularly useful for transforming data in a list comprehensively.

```python
def square(x):
    return x*x

square(10)
```

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

list(map(square,numbers))
```

```python
## Lambda function with map
numbers=[1,2,3,4,5,6,7,8]
list(map(lambda x:x*x,numbers))
```

```
### MAp multiple iterables

numbers1=[1,2,3]
numbers2=[4,5,6]

added_numbers=list(map(lambda x,y:x+y,numbers1,numbers2))
print(added_numbers)
```

```
c = []
for i in range(len(numbers1)):
    c.append(numbers1[i] + numbers2[i])
```

```
c
```

```
## map() to convert a list of strings to integers
# Use map to convert strings to integers
str_numbers = ['1', '2', '3', '4', '5']
int_numbers = list(map(int, str_numbers))

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

```
# a = list(map(int, input(numbers).split(',')))
```

```
words=['apple','banana','cherry']
upper_word=list(map(str.upper,words))
print(upper_word)
```

```
def get_name(person):
    return person['name']

people=[
    {'name':'Mukesh','age':29},
    {'name':'raj','age':27}
]
list(map(get_name,people))
```

```
def max(a,b):
  if a > b: return a
  else: return b
```

```
list1= [1,1,1]

list2= [0,0,0,1,1,1,1,1,1,1]

result = list(map(max,list1,list2))

print(result)
```

The filter() Function in Python

The filter() function constructs an iterator from elements of an iterable for which a function returns true. It is used to filter out items from a list (or any other iterable) based on a condition.

```
def even(num):
    if num%2==0:
        return True
```

```
even(24)
```

```
lst=[1,2,3,4,5,6,7,8,9,10,11,12]

list(filter(even,lst))
```

```
## filter with a Lambda Function
numbers=[1,2,3,4,5,6,7,8,9]
greater_than_five=list(filter(lambda x:x>5,numbers))
print(greater_than_five)
```

```
## Filter with a lambda function and multiple conditions
numbers=[1,2,3,4,5,6,7,8,9]
even_and_greater_than_five=list(filter(lambda x:x>5 and x%2==0,numbers))
print(even_and_greater_than_five)
```

```
## Filter() to check if the age is greate than 25 in dictionaries
people=[
    {'name':'Mukesh','age':32},
    {'name':'Raj','age':33},
    {'name':'Arul','age':25}
]
```

```python
def age_greater_than_25(person):
    return person['age']>25

list(filter(age_greater_than_25,people))
```

In Python, modules and packages are essential for organizing and structuring code, making it more reusable, maintainable, and easier to navigate.

1. Modules

A **module** is simply a file containing Python code. Modules can define functions, classes, variables, and runnable code. By dividing code into modules, you can avoid repetition and make your codebase more manageable.

Creating and Using Modules

Suppose you have a file called `math_operations.py` with the following code:

```python
# math_operations.py

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

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

To use this module in another Python file, you would import it as follows:

```python
import math_operations

result = math_operations.add(5, 3)
print(result)  # Output: 8
```

Importing Specific Functions or Variables

If you only need a specific function, you can use `from`:

```python
from math_operations import add

result = add(5, 3)
print(result)  # Output: 8
```

2. Packages

A **package** is a collection of modules organized in a directory hierarchy. It allows you to organize related modules together, like a folder structure. To create a package, you need to place an `__init__.py` file in the directory. This file can be empty, but it signals to Python that the directory should be treated as a package.

Creating a Package

Suppose you have a directory structure as follows:

```
my_package/
    __init__.py
    math_operations.py
    string_operations.py
```

- **`__init__.py`**: Makes `my_package` a package.
- **`math_operations.py`** and **`string_operations.py`**: Contain related functions for math and string manipulations, respectively.

Using a Package

To import a module from a package, you use dot notation:

```python
from my_package import math_operations

result = math_operations.add(5, 3)
print(result)  # Output: 8
```

Or, to access another module:

```python
from my_package import string_operations
```

Importing All Modules in a Package

If you want to make all modules available when the package is imported, you can modify `__init__.py`:

```python
# my_package/__init__.py
from .math_operations import add, subtract
from .string_operations import concatenate, split
```

Now, when you import the package, all specified modules are available:

```
from my_package import add, concatenate
```

3. Relative Imports

Within packages, you can use relative imports to import modules. For example, in `string_operations.py`, to import something from `math_operations.py`:

```
from .math_operations import add
```

4. Best Practices

- Use packages for better organization, especially for larger projects.
- Use descriptive names for modules and packages.
- Avoid wildcard imports (`from module import *`) for better readability and to prevent namespace pollution.
- Utilize `__all__` in `__init__.py` for controlling the public API of a package.

This modular approach helps maintain organized and reusable code!

Initializing Packages

1. The `__init__.py` file can be used to initialize a package and can also be used to define which modules should be exposed when a package is imported.

2. If the `__init__.py file is empty`, it only tells Python that the directory should be treated as a package. You can include initialization code and control what is exposed when using `from package_name import * by defining the __all__ list`.

```
# package_name/__init__.py
__all__ = ['module1', 'module2']
```

Manipulating Strings

Escape characters

An escape character is created by typing a backslash \ followed by the character you want to insert.

Escape character	Prints as
`\'`	Single quote
`\"`	Double quote
`\t`	Tab
`\n`	Newline (line break)
`\\`	Backslash
`\b`	Backspace
`\ooo`	Octal value
`\r`	Carriage Return

```
>>> print("Hello there!\nHow are you?\nI\'m doing fine.")
# Hello there!
# How are you?
# I'm doing fine.
```

Raw strings

A raw string entirely ignores all escape characters and prints any backslash that appears in the string.

```
>>> print(r"Hello there!\nHow are you?\nI\'m doing fine.")
# Hello there!\nHow are you?\nI\'m doing fine.
```

Raw strings are mostly used for regular expression definition.

Multiline Strings

```
>>> print(
... """Dear Alice,
...
... Eve's cat has been arrested for catnapping,
... cat burglary, and extortion.
...
... Sincerely,
... Bob"""
... )

# Dear Alice,
# Eve's cat has been arrested for catnapping,
# cat burglary, and extortion.

# Sincerely,
# Bob
```

Indexing and Slicing strings

```
H   e   l   l   o       w   o   r   l   d    !
0   1   2   3   4   5   6   7   8   9   10   11
```

Indexing

```
>>> spam = 'Hello world!'

>>> spam[0]
# 'H'

>>> spam[4]
# 'o'

>>> spam[-1]
# '!'
```

Slicing

```
>>> spam = 'Hello world!'

>>> spam[0:5]
# 'Hello'

>>> spam[:5]
# 'Hello'

>>> spam[6:]
# 'world!'
```

```
>>> spam[6:-1]
# 'world'

>>> spam[:-1]
# 'Hello world'

>>> spam[::-1]
# '!dlrow olleH'

>>> fizz = spam[0:5]
>>> fizz
# 'Hello'
```

The in and not in operators

```
>>> 'Hello' in 'Hello World'
# True

>>> 'Hello' in 'Hello'
# True

>>> 'HELLO' in 'Hello World'
# False

>>> '' in 'spam'
# True

>>> 'cats' not in 'cats and dogs'
# False
```

upper(), lower() and title()

Transforms a string to upper, lower and title case:

```
>>> greet = 'Hello world!'
>>> greet.upper()
# 'HELLO WORLD!'

>>> greet.lower()
```

```
# 'hello world!'

>>> greet.title()
# 'Hello World!'
```

isupper() and islower() methods

Returns `True` or `False` after evaluating if a string is in upper or lower case:

```
>>> spam = 'Hello world!'
>>> spam.islower()
# False

>>> spam.isupper()
# False

>>> 'HELLO'.isupper()
# True

>>> 'abc12345'.islower()
# True

>>> '12345'.islower()
# False

>>> '12345'.isupper()
# False
```

The isX string methods

Method	Description
isalpha()	returns `True` if the string consists only of letters.
isalnum()	returns `True` if the string consists only of letters and numbers.
isdecimal()	returns `True` if the string consists only of numbers.
isspace()	returns `True` if the string consists only of spaces, tabs, and new-lines.
istitle()	returns `True` if the string consists only of words that begin with an uppercase letter followed by only lowercase characters.

startswith() and endswith()

```
>>> 'Hello world!'.startswith('Hello')
# True

>>> 'Hello world!'.endswith('world!')
# True

>>> 'abc123'.startswith('abcdef')
# False

>>> 'abc123'.endswith('12')
# False

>>> 'Hello world!'.startswith('Hello world!')
# True

>>> 'Hello world!'.endswith('Hello world!')
# True
```

join() and split()

join()

The `join()` method takes all the items in an iterable, like a list, dictionary, tuple or set, and joins them into a string. You can also specify a separator.

```
>>> ''.join(['My', 'name', 'is', 'Simon'])
'MynameisSimon'

>>> ', '.join(['cats', 'rats', 'bats'])
# 'cats, rats, bats'

>>> ' '.join(['My', 'name', 'is', 'Simon'])
# 'My name is Simon'

>>> 'ABC'.join(['My', 'name', 'is', 'Simon'])
# 'MyABCnameABCisABCSimon'
```

split()

The `split()` method splits a `string` into a `list`. By default, it will use whitespace to separate the items, but you can also set another character of choice:

```
>>> 'My name is Simon'.split()
# ['My', 'name', 'is', 'Simon']

>>> 'MyABCnameABCisABCSimon'.split('ABC')
# ['My', 'name', 'is', 'Simon']

>>> 'My name is Simon'.split('m')
# ['My na', 'e is Si', 'on']

>>> ' My  name is  Simon'.split()
# ['My', 'name', 'is', 'Simon']

>>> ' My  name is  Simon'.split(' ')
# ['', 'My', '', 'name', 'is', '', 'Simon']
```

Justifying text with rjust(), ljust() and center()

```
>>> 'Hello'.rjust(10)
# '     Hello'

>>> 'Hello'.rjust(20)
# '               Hello'

>>> 'Hello World'.rjust(20)
# '         Hello World'

>>> 'Hello'.ljust(10)
# 'Hello     '

>>> 'Hello'.center(20)
# '       Hello       '
```

An optional second argument to `rjust()` and `ljust()` will specify a fill character apart from a space character:

```
>>> 'Hello'.rjust(20, '*')
# '***************Hello'

>>> 'Hello'.ljust(20, '-')
# 'Hello---------------'

>>> 'Hello'.center(20, '=')
# '=======Hello========'
```

Removing whitespace with strip(), rstrip(), and lstrip()

```
>>> spam = '    Hello World     '
>>> spam.strip()
# 'Hello World'

>>> spam.lstrip()
# 'Hello World     '

>>> spam.rstrip()
# '    Hello World'

>>> spam = 'SpamSpamBaconSpamEggsSpamSpam'
>>> spam.strip('ampS')
# 'BaconSpamEggs'
```

The Count Method

Counts the number of occurrences of a given character or substring in the string it is applied to. Can be optionally provided start and end index.

```
>>> sentence = 'one sheep two sheep three sheep four'
>>> sentence.count('sheep')
# 3

>>> sentence.count('e')
# 9

>>> sentence.count('e', 6)
# 8
# returns count of e after 'one sh' i.e 6 chars since beginning of string

>>> sentence.count('e', 7)
# 7
```

Replace Method

Replaces all occurences of a given substring with another substring. Can be optionally provided a third argument to limit the number of replacements. Returns a new string.

```
>>> text = "Hello, world!"
>>> text.replace("world", "planet")
# 'Hello, planet!'

>>> fruits = "apple, banana, cherry, apple"
>>> fruits.replace("apple", "orange", 1)
#Optionally, the maximum number of replacements you want to make
# (1 in this case).
# 'orange, banana, cherry, apple'

>>> sentence = "I like apples, Apples are my favorite fruit"
>>> sentence.replace("apples", "oranges")
# 'I like oranges, Apples are my favorite fruit'
```

Python String Formatting

From the Python 3 documentation The formatting operations described here (% operator) exhibit a variety of quirks that lead to a number of common errors […]. Using the newer formatted string literals […] helps avoid these errors. These alternatives also provide more powerful, flexible and extensible approaches to formatting text.

% operator

Prefer String Literals For new code, using str.format, or formatted string literals (Python 3.6+) over the % operator is strongly recommended.

```
>>> name = 'Pete'
>>> 'Hello %s' % name
# "Hello Pete"
```

We can use the `%d` format specifier to convert an int value to a string:

```
>>> num = 5
>>> 'I have %d apples' % num
# "I have 5 apples"
```

str.format

Python 3 introduced a new way to do string formatting that was later back-ported to Python 2.7. This makes the syntax for string formatting more regular.

```
>>> name = 'John'
>>> age = 20

>>> "Hello I'm {}, my age is {}".format(name, age)
# "Hello I'm John, my age is 20"

>>> "Hello I'm {0}, my age is {1}".format(name, age)
# "Hello I'm John, my age is 20"
```

Formatted String Literals or f-Strings

If your are using Python 3.6+, string `f-Strings` are the recommended way to format strings.

From the Python 3 documentation A formatted string literal or f-string is a string literal that is prefixed with `f` or `F`. These strings may contain replacement fields, which are expressions delimited by curly braces {}. While other string literals always have a constant value, formatted strings are really expressions evaluated at run time.

```
>>> name = 'Elizabeth'
>>> f'Hello {name}!'
# 'Hello Elizabeth!'
```

It is even possible to do inline arithmetic with it:

```
>>> a = 5
>>> b = 10
>>> f'Five plus ten is {a + b} and not {2 * (a + b)}.'
# 'Five plus ten is 15 and not 30.'
```

Multiline f-Strings

```
>>> name = 'Robert'
>>> messages = 12
>>> (
... f'Hi, {name}. '
... f'You have {messages} unread messages'
... )
# 'Hi, Robert. You have 12 unread messages'
```

The = specifier

This will print the expression and its value:

```
>>> from datetime import datetime
>>> now = datetime.now().strftime("%b/%d/%Y - %H:%M:%S")
>>> f'date and time: {now=}'
# "date and time: now='Nov/14/2022 - 20:50:01'"
```

Adding spaces or characters

```
>>> f"{name.upper() = :-^20}"
# 'name.upper() = -------ROBERT-------'
>>>
>>> f"{name.upper() = :^20}"
# 'name.upper() =        ROBERT       '
>>>
>>> f"{name.upper() = :20}"
# 'name.upper() = ROBERT              '
```

Formatting Digits

Adding thousands separator

```
>>> a = 10000000
>>> f"{a:,}"
# '10,000,000'
```

Rounding

```
>>> a = 3.1415926
>>> f"{a:.2f}"
# '3.14'
```

Showing as Percentage

```
>>> a = 0.816562
>>> f"{a:.2%}"
# '81.66%'
```

Number formatting table

Number	Format	Output	description
3.1415926	{:.2f}	3.14	Format float 2 decimal places
3.1415926	{:+.2f}	+3.14	Format float 2 decimal places with sign
-1	{:+.2f}	-1.00	Format float 2 decimal places with sign
2.71828	{:.0f}	3	Format float with no decimal places
4	{:0>2d}	04	Pad number with zeros (left padding, width 2)
4	{:x<4d}	4xxx	Pad number with x's (right padding, width 4)
10	{:x<4d}	10xx	Pad number with x's (right padding, width 4)
1000000	{:,}	1,000,000	Number format with comma separator
0.35	{:.2%}	35.00%	Format percentage
1000000000	{:.2e}	1.00e+09	Exponent notation
11	{:11d}	11	Right-aligned (default, width 10)
11	{:<11d}	11	Left-aligned (width 10)
11	{:^11d}	11	Center aligned (width 10)

Template Strings

A simpler and less powerful mechanism, but it is recommended when handling strings generated by users. Due to their reduced complexity, template strings are a safer choice.

```
>>> from string import Template
>>> name = 'Elizabeth'
>>> t = Template('Hey $name!')
>>> t.substitute(name=name)
# 'Hey Elizabeth!'
```

Regular expressions (regex) are powerful tools for matching and manipulating text patterns. They use special sequences of characters to define search patterns, making it easy to find, replace, and manage text based on patterns. Let's go over the basics and some examples.

Basic Components of Regular Expressions

1. **Literal Characters**: Matches exactly the text you type.
   - Example: `cat` matches "cat" but not "Cat" (case-sensitive).
2. **Metacharacters**: Special characters with specific meanings.
   - `.` - Matches any character except newline.
   - `^` - Matches the start of a string.
   - `$` - Matches the end of a string.
   - `[]` - Matches any one of the characters inside.
   - `\` - Escapes metacharacters to match them literally.
3. **Character Classes**:
   - `[abc]` - Matches either `a`, `b`, or `c`.
   - `[a-z]` - Matches any lowercase letter.
   - `\d` - Matches any digit (equivalent to `[0-9]`).
   - `\D` - Matches any non-digit character.
   - `\w` - Matches any word character (alphanumeric + `_`).
   - `\W` - Matches any non-word character.
4. **Quantifiers**: Specify how many times a character or group should appear.
   - `*` - Matches 0 or more times.
   - `+` - Matches 1 or more times.
   - `?` - Matches 0 or 1 time (optional).
   - `{n}` - Matches exactly `n` times.
   - `{n,}` - Matches `n` or more times.
   - `{n,m}` - Matches between `n` and `m` times.
5. **Groups and Alternation**:
   - `( )` - Groups part of the regex. Useful for capturing.
   - `|` - Logical OR operator, matching either pattern.
6. **Anchors**:
   - `^` - Matches the start of a line.
   - `$` - Matches the end of a line.
   - Example: `^cat$` will match only if the string is exactly "cat".

Examples

Example 1: Matching a Simple Email Pattern

- Regex: `\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,7}\b`
- Explanation:
  - `[A-Za-z0-9._%+-]+`: Matches the username part of an email (alphanumeric characters and allowed symbols).
  - `@`: Matches the `@` symbol.
  - `[A-Za-z0-9.-]+`: Matches the domain part of the email.
  - `\.`: Escapes the dot, matching a literal `.`.
  - `[A-Z|a-z]{2,7}`: Matches the top-level domain (e.g., `com`, `org`).

Example 2: Extracting a Phone Number

- Regex: `(\d{3})-(\d{3})-(\d{4})`
- Explanation:
  - `(\d{3})`: Matches the area code (3 digits).
  - `-`: Matches the hyphen.
  - `(\d{3})` and `(\d{4})`: Match the next 3 and 4 digits of the phone number.

Example 3: Validating a Strong Password

- Regex: `^(?=.*[A-Za-z])(?=.*\d)(?=.*[@$!%*#?&])[A-Za-z\d@$!%*#?&]{8,}$`
- Explanation:
  - `(?=.*[A-Za-z])`: Ensures at least one letter.
  - `(?=.*\d)`: Ensures at least one digit.
  - `(?=.*[@$!%*#?&])`: Ensures at least one special character.
  - `[A-Za-z\d@$!%*#?&]{8,}`: Ensures the password is at least 8 characters long.

Regular Expression Functions in Python

In Python, the `re` module provides various functions for regex:

1. `re.search()`: Searches for the first match of the pattern in the string.
2. `re.match()`: Checks if the beginning of the string matches the pattern.
3. `re.findall()`: Returns a list of all non-overlapping matches in the string.
4. `re.sub()`: Replaces matches in the string with a given replacement.

Practical Examples with Code

```
import re

# Example 1: Email
email_pattern = r"\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,7}\b"
text = "My email is example@test.com"
print(re.findall(email_pattern, text))

# Example 2: Phone Number
phone_pattern = r"(\d{3})-(\d{3})-(\d{4})"
phone_text = "Call me at 123-456-7890"
print(re.findall(phone_pattern, phone_text))

# Example 3: Password Validation
pattern = r"^(?=.*[A-Za-z])(?=.*\d)(?=.*[@$!%*#?&])[A-Za-z\d@$!%*#?&]{8,}$"
password = "StrongPass@123"
print(bool(re.match(pattern, password)))  # True if it matches
```

Summary

Regular expressions are highly customizable and can be complex, but mastering the basics—characters, metacharacters, quantifiers, and functions—will make it much easier to search and manipulate text efficiently. They're especially useful in parsing, data cleaning, and validation.

Let's dive into regular expressions in Python with the `re` module, using simple examples to demonstrate common regex concepts.

1. Importing the `re` Module

To use regular expressions in Python, you first need to import the `re` module:

```
import re
```

2. Using Basic `re` Functions

Python's `re` module provides several functions for regex-based operations:

- **`re.search()`** - Searches for the first match of the pattern in a string.

- `re.match()` - Checks if the pattern matches the beginning of the string.
- `re.findall()` - Returns all occurrences of the pattern.
- `re.sub()` - Replaces occurrences of a pattern with a specified string.

Let's explore each of these with examples.

Example 1: Simple Match with `re.match()`

Task: Check if a string starts with "Hello."

```
pattern = r"^Hello"
text = "Hello, world!"

# Using re.match
if re.match(pattern, text):
    print("Match found!")
else:
    print("No match.")
```

Explanation: - `^Hello` - The ^ asserts that "Hello" should be at the start of the string. - `re.match()` checks only at the beginning of the string.

Example 2: Searching for a Pattern Anywhere in the String with `re.search()`

Task: Find if a phone number is in the format "123-456-7890."

```
pattern = r"\d{3}-\d{3}-\d{4}"
text = "My phone number is 123-456-7890."

# Using re.search
match = re.search(pattern, text)
if match:
    print("Phone number found:", match.group())
else:
    print("No phone number found.")
```

Explanation: - `\d{3}-\d{3}-\d{4}` - Matches a pattern of three digits, a hyphen, three digits, another hyphen, and four digits. - `re.search()` scans the entire string for a match, returning the first occurrence if found.

Example 3: Finding All Occurrences with `re.findall()`

Task: Extract all the email addresses in a string.

```
pattern = r"\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,7}\b"
text = "Contact us at support@example.com or sales@example.org."

# Using re.findall
emails = re.findall(pattern, text)
print("Emails found:", emails)
```

Explanation: - `\b...` - The `\b` boundary ensures it matches complete words. - `re.findall()` returns all matches in a list, which is useful for extracting multiple occurrences.

Example 4: Replacing Text with `re.sub()`

Task: Replace all digits in a string with an asterisk `*`.

```
pattern = r"\d"
text = "My pin code is 12345 and my ID is 67890."

# Using re.sub
masked_text = re.sub(pattern, "*", text)
print("Masked text:", masked_text)
```

Explanation: - `\d` - Matches any digit. - `re.sub()` replaces each match (each digit here) with `*`.

Example 5: Validating with Regular Expressions

Task: Check if a password is strong (at least 8 characters, includes a letter, a number, and a special character).

```
pattern = r"^(?=.*[A-Za-z])(?=.*\d)(?=.*[@$!%*#?&])[A-Za-z\d@$!%*#?&]{8,}$"
password = "StrongPass@123"

# Using re.match
if re.match(pattern, password):
    print("Password is strong.")
else:
    print("Password is weak.")
```

Explanation: - `(?=.*[A-Za-z])` - Asserts there is at least one letter. - `(?=.*\d)` - Asserts there is at least one digit. - `(?=.*[@$!%*#?&])` - Asserts there is at least one special character. - `{8,}` - Specifies that the length must be at least 8 characters.

Full Summary Code

Here's a summary of all examples in one place:

```
import re

# 1. Simple Match
print("\n1. Simple Match:")
pattern = r"^Hello"
text = "Hello, world!"
print("Match found!" if re.match(pattern, text) else "No match.")

# 2. Searching Anywhere
print("\n2. Searching for a Phone Number:")
pattern = r"\d{3}-\d{3}-\d{4}"
text = "My phone number is 123-456-7890."
match = re.search(pattern, text)
print("Phone number found:", match.group() if match else "Not found.")

# 3. Finding All Occurrences
print("\n3. Finding All Emails:")
```

```
pattern = r"\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,7}\b"
text = "Contact us at support@example.com or sales@example.org."
print("Emails found:", re.findall(pattern, text))

# 4. Replacing Text
print("\n4. Replacing Digits with '*':")
pattern = r"\d"
text = "My pin code is 12345 and my ID is 67890."
print("Masked text:", re.sub(pattern, "*", text))

# 5. Validating a Strong Password
print("\n5. Validating a Password:")
pattern = r"^(?=.*[A-Za-z])(?=.*\d)(?=.*[@$!%*#?&])[A-Za-z\d@$!%*#?&]{8,}$"
password = "StrongPass@123"
print("Strong_Password." if re.match(pattern, password) else "Weak_Password.")
```

This code demonstrates various regex tasks, like matching, searching, finding, replacing, and validating patterns, to help with text processing and data validation.

Regular expression **flags** in Python modify the behavior of regex patterns. They're useful for controlling case sensitivity, multi-line processing, and more. In Python, flags are specified as optional parameters in `re` functions, or added to the pattern with a `(?i)` syntax.

Here's a list of the most common flags:

Common Regex Flags

1. **`re.IGNORECASE` or `re.I`** - Makes the regex case-insensitive.
2. **`re.MULTILINE` or `re.M`** - Allows `^` and `$` to match the start and end of each line, rather than the entire string.
3. **`re.DOTALL` or `re.S`** - Allows the dot `.` to match any character, including newlines.
4. **`re.VERBOSE` or `re.X`** - Allows whitespace and comments in the pattern, improving readability.
5. **`re.ASCII` or `re.A`** - Interprets `\w`, `\d`, `\s`, and similar as ASCII-only, rather than Unicode.

Examples Using Flags

Let's go over each of these with code examples.

---

1. `re.IGNORECASE` (Case-Insensitive Matching)

This flag makes the pattern case-insensitive, so it matches both uppercase and lowercase characters.

```python
import re

pattern = r"hello"
text = "Hello, world!"

# Using re.IGNORECASE flag
match = re.search(pattern, text, re.IGNORECASE)
print("Match found!" if match else "No match.")  # Output: Match found!
```

Explanation: - `re.IGNORECASE` allows `hello` to match `Hello`, ignoring case.

---

2. `re.MULTILINE` (Multi-Line Matching)

This flag changes the behavior of `^` and `$` to match at the start and end of each line, not just the beginning and end of the entire string.

```python
pattern = r"^hello"
text = "hello\nworld\nhello"

# Using re.MULTILINE flag
matches = re.findall(pattern, text, re.MULTILINE)
print("Matches found:", matches)  # Output: ['hello', 'hello']
```

Explanation: - With `re.MULTILINE`, `^hello` matches `hello` at the start of each line.

---

3. `re.DOTALL` (Dot Matches Newline)

The `.` character usually matches any character except newlines. Using `re.DOTALL` makes `.` match any character, including newline characters.

```
pattern = r"hello.*world"
text = "hello\nworld"

# Using re.DOTALL flag
match = re.search(pattern, text, re.DOTALL)
print("Match found!" if match else "No match.")  # Output: Match found!
```

Explanation: - With `re.DOTALL`, `.*` can span across lines, so it matches `hello` followed by `world` on the next line.

---

4. `re.VERBOSE` (Readable Regular Expressions)

The `re.VERBOSE` flag allows for whitespace and comments in the pattern, which makes complex regex more readable.

```
pattern = r"""
    ^               # Start of the string
    [A-Za-z0-9._%+-]+  # Username part
    @               # At symbol
    [A-Za-z0-9.-]+     # Domain part
    \.              # Dot
    [A-Za-z]{2,7}   # TLD (e.g., com, org, net)
    $               # End of the string
"""
text = "contact@example.com"

# Using re.VERBOSE flag
match = re.match(pattern, text, re.VERBOSE)
print("Match found!" if match else "No match.")  # Output: Match found!
```

Explanation: - `re.VERBOSE` allows whitespace and comments, improving readability without affecting the regex's functionality.

---

5. `re.ASCII` (ASCII-Only Matching)

The `re.ASCII` flag forces regex patterns to match only ASCII characters when using special sequences like `\w`, `\d`, or `\s`.

```
pattern = r"\w+"
text = "Café"

# Without re.ASCII
print("Without re.ASCII:", re.findall(pattern, text))# Output: ['Café']

# With re.ASCII
print("With re.ASCII:", re.findall(pattern, text, re.ASCII))# Output: ['Caf']
```

Explanation: - Without `re.ASCII`, `\w` includes Unicode characters (like `é`). With `re.ASCII`, only ASCII characters are matched.

Combining Flags

You can combine multiple flags using the `|` (bitwise OR) operator.

```
pattern = r"^hello.*world"
text = "Hello\nworld"

# Combining re.IGNORECASE and re.DOTALL
match = re.search(pattern, text, re.IGNORECASE | re.DOTALL)
print("Match found!" if match else "No match.")  # Output: Match found!
```

Explanation: - Here, `re.IGNORECASE` makes the search case-insensitive, and `re.DOTALL` allows `.*` to match across lines.

Python file handling is a fundamental part of the language, enabling us to create, read, update, and delete files directly. Let's dive into everything from the basics to more advanced techniques.

1. Basic File Operations

In Python, the `open()` function is used to open a file. It has the following syntax:

```
file_object = open("filename", "mode")
```

The `mode` specifies the operation to be performed on the file: - `'r'` - Read (default mode) - `'w'` - Write (overwrites the file) - `'a'` - Append (adds data to the end of the file) - `'x'` - Create a new file and open it for writing - `'b'` - Binary mode - `'t'` - Text mode (default)

Common Modes:

- `'rt'`: Read text file (default)
- `'rb'`: Read binary file
- `'wt'`: Write text file (overwrites)
- `'wb'`: Write binary file
- `'at'`: Append text file
- `'ab'`: Append binary file

2. Reading from a File

Python provides several methods to read files:

2.1 `read()` method

Reads the entire file.

```
with open("example.txt", "r") as file:
    content = file.read()
print(content)
```

2.2 `readline()` method

Reads one line at a time, useful for looping through lines in large files.

```
with open("example.txt", "r") as file:
    line = file.readline()
    while line:
        print(line.strip())
        line = file.readline()
```

2.3 `readlines()` method

Reads all lines and returns them as a list.

```
with open("example.txt", "r") as file:
    lines = file.readlines()
for line in lines:
    print(line.strip())
```

3. Writing to a File

Writing to files involves different techniques based on whether you want to overwrite the file or append to it.

3.1 `write()` method

Writes a single line or string to a file.

```
with open("example.txt", "w") as file:
    file.write("Hello, World!")
```

3.2 `writelines()` method

Writes a list of strings to the file.

```
lines = ["Line 1\n", "Line 2\n", "Line 3\n"]
with open("example.txt", "w") as file:
    file.writelines(lines)
```

4. Appending to a File

Use the `a` mode to add to the file without overwriting existing content.

```
with open("example.txt", "a") as file:
    file.write("Appending this line.")
```

5. Binary File Handling

Files like images, videos, or audio are binary files, and they require binary mode for reading or writing.

```
# Reading a binary file
with open("image.jpg", "rb") as binary_file:
    binary_data = binary_file.read()

# Writing a binary file
with open("new_image.jpg", "wb") as binary_file:
    binary_file.write(binary_data)
```

6. File Positioning with `seek()` and `tell()`

- **`seek(offset, whence)`**: Moves the file cursor to a specific position.
  - `offset`: Number of bytes to move.
  - `whence`: Position to start from (`0` = start, `1` = current, `2` = end).

```
with open("example.txt", "r") as file:
    file.seek(5)  # Move cursor to the 5th byte
    content = file.read()
    print(content)
```

- **`tell()`**: Returns the current position of the cursor.

```
with open("example.txt", "r") as file:
    file.read(5)
    print(file.tell())  # Prints position after reading 5 bytes
```

7. File Existence Check and Deletion

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

```
import os

# Check if file exists
if os.path.exists("example.txt"):
    print("File exists.")
else:
```

```python
    print("File does not exist.")

# Delete a file
if os.path.exists("example.txt"):
    os.remove("example.txt")
```

8. Using Context Manager (`with` Statement)

Using `with` is a best practice, as it ensures files are properly closed after operations, even if exceptions occur.

```python
with open("example.txt", "r") as file:
    content = file.read()
```

9. Working with JSON Files

JSON files are commonly used for structured data. The `json` module helps read and write JSON data.

```python
import json

# Writing JSON
data = {"name": "Alice", "age": 25}
with open("data.json", "w") as file:
    json.dump(data, file)

# Reading JSON
with open("data.json", "r") as file:
    data = json.load(file)
print(data)
```

10. Working with CSV Files

For handling CSV files, the `csv` module provides an easy interface.

```python
import csv

# Writing CSV
data = [["Name", "Age"], ["Alice", 25], ["Bob", 30]]
```

```python
with open("data.csv", "w", newline="") as file:
    writer = csv.writer(file)
    writer.writerows(data)

# Reading CSV
with open("data.csv", "r") as file:
    reader = csv.reader(file)
    for row in reader:
        print(row)
```

11. File Compression with `gzip` and `zipfile`

11.1 `gzip` (single file compression)

```python
import gzip

# Writing compressed file
with gzip.open("file.txt.gz", "wb") as gz_file:
    gz_file.write(b"This is some text data.")

# Reading compressed file
with gzip.open("file.txt.gz", "rb") as gz_file:
    content = gz_file.read()
    print(content)
```

11.2 `zipfile` (multiple files in one archive)

```python
import zipfile

# Creating a zip file
with zipfile.ZipFile("files.zip", "w") as zip_file:
    zip_file.write("file1.txt")
    zip_file.write("file2.txt")

# Extracting a zip file
with zipfile.ZipFile("files.zip", "r") as zip_file:
    zip_file.extractall("extracted_files")
```

12. File Permissions and Access Control

Using the `os` module, you can check and set permissions:

```
import os

# Check permissions
print(oct(os.stat("example.txt").st_mode)[-3:])

# Set permissions (e.g., read-only for all)
os.chmod("example.txt", 0o444)
```

13. Reading Files in a Memory-Efficient Way

If you're dealing with large files, you can process them line-by-line without loading the entire file into memory:

```
with open("large_file.txt", "r") as file:
    for line in file:
        print(line.strip())
```

14. Error Handling in File Operations

To handle potential file errors, use a try-except block:

```
try:
    with open("example.txt", "r") as file:
        content = file.read()
except FileNotFoundError:
    print("File not found.")
except IOError:
    print("IO error occurred.")
```

15. Encoding and Decoding

Python's `open()` allows you to specify the encoding of the file, which is helpful for non-ASCII text.

```
with open("file.txt", "r", encoding="utf-8") as file:
    content = file.read()
print(content)
```

These are the most comprehensive aspects of file handling in Python! With this knowledge, you can effectively work with various file types and manage your file I/O tasks.

Python Exception Handling

In computing and computer programming, exception handling is the process of responding to the occurrence of exceptions – anomalous or exceptional conditions requiring special processing.

Python has many built-in exceptions that are raised when a program encounters an error, and most external libraries, like the popular Requests, include his own custom exceptions that we will need to deal to.

Basic exception handling

You can't divide by zero, that is a mathematical true, and if you try to do it in Python, the interpreter will raise the built-in exception ZeroDivisionError:

```
>>> def divide(dividend , divisor):
...     print(dividend / divisor)
...
>>> divide(dividend=10, divisor=5)
# 5

>>> divide(dividend=10, divisor=0)
# Traceback (most recent call last):
#   File "<stdin>", line 1, in <module>
# ZeroDivisionError: division by zero
```

Let's say we don't want our program to stop its execution or show the user an output he will not understand. Say we want to print a useful and clear message, then we need to ***handle*** the exception with the `try` and `except` keywords:

```
>>> def divide(dividend , divisor):
...     try:
...         print(dividend / divisor)
...     except ZeroDivisionError:
...         print('You can not divide by 0')
...
>>> divide(dividend=10, divisor=5)
# 5

>>> divide(dividend=10, divisor=0)
# You can not divide by 0
```

Handling Multiple exceptions using one exception block

You can also handle multiple exceptions in one line like the following without the need to create multiple exception blocks.

```
>>> def divide(dividend , divisor):
...     try:
...         var = 'str' + 1
...         print(dividend / divisor)
...     except (ZeroDivisionError, TypeError) as error:
...         print(error)
...
>>> divide(dividend=10, divisor=5)
# 5

>>> divide(dividend=10, divisor=0)
# `division by zero` Error message
# `can only concatenate str (not "int") to str` Error message
```

Finally code in exception handling

The code inside the `finally` section is always executed, no matter if an exception has been raised or not:

```
>>> def divide(dividend , divisor):
...     try:
...         print(dividend / divisor)
...     except ZeroDivisionError:
```

```
...         print('You can not divide by 0')
...     finally:
...         print('Execution finished')
...
>>> divide(dividend=10, divisor=5)
# 5
# Execution finished

>>> divide(dividend=10, divisor=0)
# You can not divide by 0
# Execution finished
```

Custom Exceptions

Custom exceptions initialize by creating a `class` that inherits from the base `Exception` class of Python, and are raised using the `raise` keyword:

```
>>> class MyCustomException(Exception):
...     pass
...
>>> raise MyCustomException
# Traceback (most recent call last):
#   File "<stdin>", line 1, in <module>
# __main__.MyCustomException
```

To declare a custom exception message, you can pass it as a parameter:

```
>>> class MyCustomException(Exception):
...     pass
...
>>> raise MyCustomException('A custom message for my custom exception')
# Traceback (most recent call last):
#   File "<stdin>", line 1, in <module>
# __main__.MyCustomException: A custom message for my custom exception
```

Handling a custom exception is the same as any other:

```
>>> try:
...     raise MyCustomException('A custom message for my custom exception')
>>> except MyCustomException:
```

```
...     print('My custom exception was raised')
...
# My custom exception was raised
```

Python Debugging

Finding and resolving bugs In computer programming and software development, debugging is the process of finding and resolving bugs (defects or problems that prevent correct operation) within computer programs, software, or systems.

Raising Exceptions

Exceptions are raised with a raise statement. In code, a raise statement consists of the following:

- The `raise` keyword
- A call to the `Exception()` function
- A string with a helpful error message passed to the `Exception()` function

```
>>> raise Exception('This is the error message.')
# Traceback (most recent call last):
#   File "<pyshell#191>", line 1, in <module>
#     raise Exception('This is the error message.')
# Exception: This is the error message.
```

Typically, it's the code that calls the function, not the function itself, that knows how to handle an exception. So, you will commonly see a raise statement inside a function and the `try` and `except` statements in the code calling the function.

```
>>> def box_print(symbol, width, height):
...     if len(symbol) != 1:
...       raise Exception('Symbol must be a single character string.')
...     if width <= 2:
...       raise Exception('Width must be greater than 2.')
...     if height <= 2:
...       raise Exception('Height must be greater than 2.')
...     print(symbol * width)
...     for i in range(height - 2):
...         print(symbol + (' ' * (width - 2)) + symbol)
...     print(symbol * width)
```

```
...
>>> for sym, w, h in (('*', 4, 4), ('O', 20, 5), ('x', 1, 3), ('ZZ', 3, 3)):
...     try:
...         box_print(sym, w, h)
...     except Exception as err:
...         print('An exception happened: ' + str(err))
...
# ****
# *  *
# *  *
# ****
# OOOOOOOOOOOOOOOOOOOO
# O                  O
# O                  O
# O                  O
# OOOOOOOOOOOOOOOOOOOO
# An exception happened: Width must be greater than 2.
# An exception happened: Symbol must be a single character string.
```

Read more about Exception Handling.

Getting the Traceback as a string

The `traceback` is displayed by Python whenever a raised exception goes unhandled. But can also obtain it as a string by calling traceback.format_exc(). This function is useful if you want the information from an exception's traceback but also want an except statement to gracefully handle the exception. You will need to import Python's traceback module before calling this function.

```
>>> import traceback

>>> try:
...     raise Exception('This is the error message.')
>>> except:
...     with open('errorInfo.txt', 'w') as error_file:
...         error_file.write(traceback.format_exc())
...     print('The traceback info was written to errorInfo.txt.')
...
# 116
# The traceback info was written to errorInfo.txt.
```

The 116 is the return value from the `write()` method, since 116 characters were written to the file. The `traceback` text was written to errorInfo.txt.

```
Traceback (most recent call last):
  File "<pyshell#28>", line 2, in <module>
Exception: This is the error message.
```

Assertions

An assertion is a sanity check to make sure your code isn't doing something obviously wrong. These sanity checks are performed by `assert` statements. If the sanity check fails, then an `AssertionError` exception is raised. In code, an `assert` statement consists of the following:

- The `assert` keyword
- A condition (that is, an expression that evaluates to `True` or `False`)
- A comma
- A `string` to display when the condition is `False`

The `assert` statement in Python is used for debugging and testing. It helps you test if a condition is true. If the condition is false, an `AssertionError` is raised, optionally with an error message. This is especially useful in development to catch unexpected bugs.

Syntax:

```
assert condition, "Optional error message"
```

Examples:

1. **Basic Assertion:**

   ```
   x = 5
   assert x > 0  # No error as 5 > 0
   # Raises AssertionError with message
   assert x < 0, "x should be less than 0"
   ```

2. **Checking Function Output:**

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

   result = add(3, 4)
   assert result == 7, "The result should be 7"
   ```

3. **Validating User Input:**

   ```
   age = 20
   assert age >= 18, "Age must be at least 18"
   ```

Using `assert` is a quick way to check assumptions and catch errors early in your code!

In plain English, an assert statement says, "I assert that this condition holds true, and if not, there is a bug somewhere in the program." Unlike exceptions, your code should not handle assert statements with try and except; if an assert fails, your program should crash. By failing fast like this, you shorten the time between the original cause of the bug and when you first notice the bug. This will reduce the amount of code you will have to check before finding the code that's causing the bug.

Disabling Assertions

Assertions can be disabled by passing the `-O` option when running Python.

Logging

To enable the `logging` module to display log messages on your screen as your program runs, copy the following to the top of your program:

```
>>> import logging
>>> logging.basicConfig(
    level=logging.DEBUG, format=' %(asctime)s - %(levelname)s- %(message)s'
)
```

Say you wrote a function to calculate the factorial of a number. In mathematics, factorial 4 is 1 × 2 × 3 × 4, or 24. Factorial 7 is 1 × 2 × 3 × 4 × 5 × 6 × 7, or 5,040. Open a new file editor window and enter the following code. It has a bug in it, but you will also enter several log messages to help yourself figure out what is going wrong. Save the program as factorialLog.py.

```
>>> import logging
>>> logging.basicConfig(
    level=logging.DEBUG, format=' %(asctime)s - %(levelname)s- %(message)s'
)
>>> logging.debug('Start of program')

>>> def factorial(n):
...     logging.debug('Start of factorial(%s)' % (n))
```

```
...     total = 1
...     for i in range(1, n + 1):
...         total *= i
...         logging.debug('i is ' + str(i) + ', total is ' + str(total))
...     logging.debug('End of factorial(%s)' % (n))
...     return total
...
>>> print(factorial(5))
>>> logging.debug('End of program')
# 2015-05-23 16:20:12,664 - DEBUG - Start of program
# 2015-05-23 16:20:12,664 - DEBUG - Start of factorial(5)
# 2015-05-23 16:20:12,665 - DEBUG - i is 0, total is 0
# 2015-05-23 16:20:12,668 - DEBUG - i is 1, total is 0
# 2015-05-23 16:20:12,670 - DEBUG - i is 2, total is 0
# 2015-05-23 16:20:12,673 - DEBUG - i is 3, total is 0
# 2015-05-23 16:20:12,675 - DEBUG - i is 4, total is 0
# 2015-05-23 16:20:12,678 - DEBUG - i is 5, total is 0
# 2015-05-23 16:20:12,680 - DEBUG - End of factorial(5)
# 0
# 2015-05-23 16:20:12,684 - DEBUG - End of program
```

Logging Levels

Logging levels provide a way to categorize your log messages by importance. There are five logging levels, described in Table 10-1 from least to most important. Messages can be logged at each level using a different logging function.

Level	Logging Function	Description
`DEBUG`	`logging.debug()`	The lowest level. Used for small details. Usually you care about these messages only when diagnosing problems.
`INFO`	`logging.info()`	Used to record information on general events in your program or confirm that things are working at their point in the program.
`WARNING`	`logging.warning()`	Used to indicate a potential problem that doesn't prevent the program from working but might do so in the future.
`ERROR`	`logging.error()`	Used to record an error that caused the program to fail to do something.
`CRITICAL`	`logging.critical()`	The highest level. Used to indicate a fatal error that has caused or is about to cause the program to stop running entirely.

Disabling Logging

After you've debugged your program, you probably don't want all these log messages cluttering the screen. The logging.disable() function disables these so that you don't have to go into your program and remove all the logging calls by hand.

```
>>> import logging

>>> logging.basicConfig(
    level=logging.INFO, format=' %(asctime)s -%(levelname)s - %(message)s'
)
>>> logging.critical('Critical error! Critical error!')
# 2015-05-22 11:10:48,054 - CRITICAL - Critical error! Critical error!

>>> logging.disable(logging.CRITICAL)
>>> logging.critical('Critical error! Critical error!')
>>> logging.error('Error! Error!')
```

Logging to a File

Instead of displaying the log messages to the screen, you can write them to a text file. The `logging.basicConfig()` function takes a filename keyword argument, like so:

```
>>> import logging
>>> logging.basicConfig(
    filename='myProgramLog.txt', level=logging.DEBUG, format='%(asctime)s - %(levelname)s - %
)
```

Loging

Logging in Python is a powerful way to track events that happen when software runs, which is especially useful for debugging, monitoring, and analyzing code in production. Python's built-in `logging` module provides a flexible framework for emitting log messages from your code.

Why Use Logging?

Logging offers more control than `print` statements and lets you: - Set different levels of importance for messages. - Control the format and destination of logs (e.g., console, file, remote server). - Filter messages so that only relevant logs are stored. - Allow different parts of an application to generate logs independently.

Basic Concepts of Logging

1. **Loggers**: The primary entities that emit log messages. A logger is created with `logging.getLogger(name)`, and it generates logs of different levels.
2. **Handlers**: Define where log messages go (e.g., console, file, or HTTP endpoint).
3. **Formatters**: Specify the layout and structure of log messages.
4. **Log Levels**: Define the severity of logs, allowing filtering based on importance. Levels include:
   - `DEBUG`: Detailed information, typically for diagnosing problems.
   - `INFO`: General events, like application progress.
   - `WARNING`: An indication that something unexpected happened, or a problem may occur.
   - `ERROR`: A more serious problem.
   - `CRITICAL`: A very serious error, indicating a program crash or data loss.

Getting Started with Basic Logging

You can quickly log messages using `logging.basicConfig()` and log at different levels.

```python
import logging

# Configure the basic settings for logging
logging.basicConfig(
    level=logging.DEBUG, format='%(asctime)s - %(levelname)s - %(message)s'
)

# Log messages of different severity
logging.debug("This is a debug message")
logging.info("This is an info message")
logging.warning("This is a warning message")
logging.error("This is an error message")
logging.critical("This is a critical message")
```

- **Output**:

  ```
  2024-11-07 10:00:00 - DEBUG - This is a debug message
  2024-11-07 10:00:01 - INFO - This is an info message
  ...
  ```

Configuring Handlers and Formatters

Adding Handlers

Handlers determine where the logs will go. Python has multiple built-in handlers, including: - `StreamHandler`: Logs to the console. - `FileHandler`: Logs to a file. - `RotatingFileHandler`: Logs to a file but creates a new one once the file reaches a specified size. - `TimedRotatingFileHandler`: Creates new log files at specified intervals (daily, weekly, etc.).

```
# Create a custom logger
logger = logging.getLogger("my_logger")
logger.setLevel(logging.DEBUG)

# Create handlers
console_handler = logging.StreamHandler()  # Console handler
file_handler = logging.FileHandler("logfile.log")  # File handler

# Set level for handlers (optional)
# Only warnings and above will go to console
console_handler.setLevel(logging.WARNING)
# All debug and above messages go to the file
file_handler.setLevel(logging.DEBUG)

# Add handlers to logger
logger.addHandler(console_handler)
logger.addHandler(file_handler)
```

Adding Formatters

Formatters define the output format of log messages, including details like timestamp, message level, and file location.

```
formatter = logging.Formatter(
    '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
)
console_handler.setFormatter(formatter)
file_handler.setFormatter(formatter)
```

Example of a Fully Configured Logger

```python
import logging

# Create logger
logger = logging.getLogger("app_logger")
logger.setLevel(logging.DEBUG)

# Create handlers
c_handler = logging.StreamHandler()
f_handler = logging.FileHandler("file.log")

# Set level for handlers
c_handler.setLevel(logging.WARNING)
f_handler.setLevel(logging.DEBUG)

# Create formatters and add to handlers
c_format = logging.Formatter(
    '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
)
f_format = logging.Formatter(
    '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
)

c_handler.setFormatter(c_format)
f_handler.setFormatter(f_format)

# Add handlers to logger
logger.addHandler(c_handler)
logger.addHandler(f_handler)

# Example usage
logger.debug("This is a debug message")
logger.info("This is an info message")
logger.warning("This is a warning message")
logger.error("This is an error message")
logger.critical("This is a critical message")
```

Advanced Features

Logging Exceptions

For logging exceptions with traceback, use `logger.exception()` within an `except` block.

```
try:
    1 / 0
except ZeroDivisionError:
    logger.exception("Division by zero error occurred")
```

Custom Log Levels

You can define custom log levels if your application has specific needs.

```
CUSTOM_LEVEL = 25  # Define custom level between standard ones
logging.addLevelName(CUSTOM_LEVEL, "CUSTOM")

def custom(self, message, *args, **kwargs):
    if self.isEnabledFor(CUSTOM_LEVEL):
        self._log(CUSTOM_LEVEL, message, args, **kwargs)

logging.Logger.custom = custom
logger.custom("This is a custom log level message")
```

Rotating Log Files

`RotatingFileHandler` is useful to manage disk space by limiting log file size.

```
from logging.handlers import RotatingFileHandler

rotating_handler = RotatingFileHandler(
    "app.log", maxBytes=2000, backupCount=5
)
rotating_handler.setFormatter(formatter)
logger.addHandler(rotating_handler)
```

Using Logging Configuration Files

Python supports setting up logging using configuration files (JSON, YAML, or `logging.config.dictConfig` with a dictionary).

Example using `dictConfig`

```python
import logging.config

logging_config = {
    "version": 1,
    "formatters": {
        "default": {
            "format": "%(asctime)s - %(name)s - %(levelname)s - %(message)s",
        },
    },
    "handlers": {
        "console": {
            "class": "logging.StreamHandler",
            "level": "DEBUG",
            "formatter": "default",
        },
        "file": {
            "class": "logging.FileHandler",
            "filename": "app.log",
            "level": "WARNING",
            "formatter": "default",
        },
    },
    "loggers": {
        "": {  # root logger
            "level": "DEBUG",
            "handlers": ["console", "file"],
        },
    },
}

logging.config.dictConfig(logging_config)
logger = logging.getLogger("example_logger")
logger.info("This is an info message")
```

Best Practices for Logging

1. **Set the appropriate log level** for production (`WARNING` or `ERROR`) vs. development (`DEBUG`).
2. **Avoid logging sensitive information** (like passwords or API keys).
3. **Use structured logging** for complex applications (e.g., using JSON for easier parsing).

4. **Log useful context** for debugging (e.g., request IDs, user information in web applications).
5. **Regularly archive and clean up log files** if logging to disk, using rotating file handlers if needed.

This should give you a comprehensive foundation on logging in Python!

```
## configuring logging
import logging

# logging.basicConfig(
#     filename='app.log',
#     filemode='w',
#     level=logging.DEBUG,
#     format='%(asctime)s-%(name)s-%(levelname)s-%(message)s',
#     datefmt='%Y-%m-%d %H:%M:%S'
#     )

## log messages with different severity levels
# logging.debug("This is a debug message")
# logging.info("This is an info message")
# logging.warning("This is a warning message")
# logging.error("This is an error message")
# logging.critical("This is a critical message")
```

```
import logging

## logging setting

# logging.basicConfig(
#     level=logging.DEBUG,
#     format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
#     datefmt='%Y-%m-%d %H:%M:%S',
#     handlers=[
#         logging.FileHandler("app1.log"),
#         logging.StreamHandler()
#     ]
# )

logger=logging.getLogger("ArithmethicApp")

def add(a,b):
    result=a+b
    logger.debug(f"Adding {a} + {b}= {result}")
    return result

def subtract(a, b):
    result = a - b
    logger.debug(f"Subtracting {a} - {b} = {result}")
    return result

def multiply(a, b):
    result = a * b
    logger.debug(f"Multiplying {a} * {b} = {result}")
    return result

def divide(a, b):
    try:
        result = a / b
        logger.debug(f"Dividing {a} / {b} = {result}")
        return result
    except ZeroDivisionError:
        logger.error("Division by zero error")
        return None

add(10,15)
subtract(15,10)
multiply(10,20)
divide(20,0)
```

```
#### Logging with Multiple Loggers
#You can create multiple loggers for different parts of your application.

import logging
## create a logger for module1
logger1=logging.getLogger("module1")
logger1.setLevel(logging.DEBUG)

##create a logger for module 2

logger2=logging.getLogger("module2")
logger2.setLevel(logging.WARNING)

# Configure logging settings
logging.basicConfig(
    level=logging.DEBUG,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
    datefmt='%Y-%m-%d %H:%M:%S'
)
```

```
## log message with different loggers
logger1.debug("This is debug message for module1")
logger2.warning("This is a warning message for module 2")
logger2.error("This is an error message")
```

In Python, `*args` and `**kwargs` are commonly used for function parameter handling, especially when the number of arguments is unknown or varies. They enable a function to accept variable numbers of positional and keyword arguments.

Here's a breakdown:

1. `*args`: Variable-Length Positional Arguments

- `*args` allows a function to take any number of positional arguments, which will be passed as a tuple.
- This is useful when you want a function to handle an arbitrary number of inputs without defining each parameter individually.

Example:

```
def my_function(*args):
    for arg in args:
        print(arg)

my_function(1, 2, 3)  # Output: 1 2 3
```

In this case, `args` is a tuple containing `(1, 2, 3)`.

Practical Use of `*args`

- When you want to allow for flexible input lengths in a function, like a sum function that can take any number of numbers:

  ```
  def add(*args):
      return sum(args)

  print(add(1, 2, 3, 4))  # Output: 10
  ```

2. **kwargs: Variable-Length Keyword Arguments

- **kwargs allows a function to accept any number of keyword arguments. These arguments are passed as a dictionary.
- kwargs stands for "keyword arguments," and this pattern is helpful when passing a set of named arguments to a function.

Example:

```python
def my_function(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

my_function(name="Alice", age=30, city="Wonderland")
# Output:
# name: Alice
# age: 30
# city: Wonderland
```

In this case, kwargs is a dictionary containing {'name': 'Alice', 'age': 30, 'city': 'Wonderland'}.

Practical Use of **kwargs

- It's often used when creating functions that handle configuration options or named parameters that can vary:

  ```python
  def configure_server(**settings):
      for key, value in settings.items():
          print(f"Setting {key} to {value}")

  configure_server(host="localhost", port=8080, debug=True)
  # Output:
  # Setting host to localhost
  # Setting port to 8080
  # Setting debug to True
  ```

Combining *args and **kwargs in One Function

- You can use both *args and **kwargs together to allow for both flexible positional and keyword arguments.

Example:

```python
def my_function(*args, **kwargs):
    print("Positional arguments:", args)
    print("Keyword arguments:", kwargs)

my_function(1, 2, 3, name="Alice", age=30)
# Output:
# Positional arguments: (1, 2, 3)
# Keyword arguments: {'name': 'Alice', 'age': 30}
```

Here, `args` will be `(1, 2, 3)` and `kwargs` will be `{'name': 'Alice', 'age': 30}`.

Argument Unpacking with * and **

- You can also use `*` and `**` to unpack arguments when calling a function.

Example:

```python
def greet(greeting, name):
    print(f"{greeting}, {name}!")

args = ("Hello", "Alice")
kwargs = {"greeting": "Hi", "name": "Bob"}

greet(*args)       # Output: Hello, Alice!
greet(**kwargs)    # Output: Hi, Bob!
```

Here, `*args` unpacks the tuple as positional arguments, and `**kwargs` unpacks the dictionary as keyword arguments.

Order of Parameters

When using `*args` and `**kwargs` with other parameters, they should appear in the following order: 1. Regular positional arguments 2. `*args` 3. Keyword-only arguments (if any) 4. `**kwargs`

Example:

```python
def my_function(a, b, *args, d, **kwargs):
    print(a, b)
    print(args)
    print(d)
    print(kwargs)

my_function(1, 2, 3, 4, d=5, x=6, y=7)
# Output:
# 1 2
# (3, 4)
# 5
# {'x': 6, 'y': 7}
```

Here: - `a` and `b` are regular positional arguments. - `*args` captures additional positional arguments `(3, 4)`. - `d` is a keyword-only argument. - `**kwargs` captures additional keyword arguments `{x: 6, y: 7}`.

Use Cases

- **Flexible Functions**: When designing functions that should accept varying numbers of arguments.
- **Forwarding Arguments**: Passing arguments from one function to another, especially in decorators.
- **Configuration Options**: Handling configuration or settings for functions in a clean, extensible way.

In summary, `*args` and `**kwargs` are powerful tools in Python, providing flexibility and making functions versatile, especially when dealing with unknown numbers of arguments.

main top-level script environment

1. `__name__` and the Main Script Environment

In Python, every file is a module, and each module has a special built-in variable called `__name__`. The value of `__name__` changes depending on how the module is used:

- **When a file is run directly as a script**, `__name__` is set to `"__main__"`.
- **When a file is imported as a module** into another script, `__name__` is set to the name of that module (e.g., if the file is named `myscript.py`, then `__name__` is set to `"myscript"`).

This `__name__` variable allows Python to distinguish between the main script and imported modules.

2. Why Use `if __name__ == "__main__":`?

Using `if __name__ == "__main__":` is a Pythonic way to determine whether a script is the main program or is being used as an imported module. This block allows you to:

- **Execute code only when the script is run directly**: Code inside the `if __name__ == "__main__":` block only executes when the script is the main program.
- **Prevent code from running on import**: If the script is imported, the code inside this block won't run, making it easy to use the functions and classes from the file without executing the main script logic.

Here's a practical breakdown:

```python
# example_module.py

def main_function():
    print("This is the main function.")

if __name__ == "__main__":
    main_function()
    print("Running as the main script.")
```

- When you run `example_module.py` directly, the `main_function()` is called, and `"Running as the main script."` is printed.
- When you import `example_module` in another script (e.g., `import example_module`), `main_function()` does not execute automatically, as `__name__` in `example_module.py` is not `"__main__"`.

3. Practical Use Cases of `if __name__ == "__main__":`

This structure is highly useful for:

- **Testing**: You can add test code within the `if __name__ == "__main__":` block to run tests directly in the script without affecting its functionality as a module.
- **Modularity and Reusability**: Writing code in a way that it only executes when run directly helps create reusable modules that can be imported into other projects.
- **Script and Library Hybrid**: A file can both provide utility functions/classes for import and act as a standalone script. This is often seen in command-line tools and utility scripts.

4. Real-World Example

Consider a file `calculator.py` with functions for addition and subtraction. You might want to run it as a standalone program for basic calculations and also import it elsewhere as a utility module.

```python
# calculator.py

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

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

if __name__ == "__main__":
    # Run only if this script is executed directly
    print("Calculator Script")
    x, y = 5, 3
    print(f"{x} + {y} = {add(x, y)}")
    print(f"{x} - {y} = {subtract(x, y)}")
```

When `calculator.py` is run directly, it acts as a standalone calculator. However, if you import it (e.g., `import calculator`) in another script, it won't print the calculations, but you can use `calculator.add()` and `calculator.subtract()`.

5. Summary

- **`__name__ == "__main__"`**: This block allows code to be run only when a script is executed directly.
- **Use Cases**: Ideal for running tests, building reusable modules, and creating hybrid scripts that can function as standalone programs or libraries.
- **Modularity**: Supports the design of modular, reusable code that can easily be used in larger projects.

This pattern is a cornerstone of writing clean, modular Python code, especially in projects with multiple files and complex dependencies. It lets you structure code for both standalone usage and modular integration, aligning well with Python's philosophy of simplicity and readability.

Object-Oriented Programming (OOP)

Object-Oriented Programming (OOP) in Python is a programming paradigm based on the concept of “objects,” which can contain data in the form of attributes (variables) and code in the form of methods (functions). Python supports OOP principles, making it easier to organize code, reduce redundancy, and implement reusable components.

Here’s a detailed overview of OOP in Python:

---

1. Basic Concepts in OOP

1.1 Classes and Objects

- **Class**: A blueprint for creating objects. It defines a set of attributes and methods that the objects created from the class can have.
- **Object**: An instance of a class. Each object has its own unique set of attribute values.

```
class Dog:
    def __init__(self, name, breed):
        self.name = name  # Attribute
        self.breed = breed  # Attribute

my_dog = Dog("Buddy", "Golden Retriever")  # Creating an object
print(my_dog.name)  # Output: Buddy
```

1.2 Attributes and Methods

- **Attributes**: Variables that hold data for an object. Defined in the `__init__` method or directly within the class.
- **Methods**: Functions defined inside a class that describe the behaviors of the objects.

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

    def bark(self):  # Method
        return "Woof!"
```

```
my_dog = Dog("Buddy", "Golden Retriever")
print(my_dog.bark())  # Output: Woof!
```

---

2. Four Pillars of OOP

2.1 Encapsulation

Encapsulation restricts access to the inner workings of an object and protects its data from unintended modification. In Python, this is achieved by defining private attributes and methods with an underscore (_) or double underscore (__) prefix.

```
class Account:
    def __init__(self, balance):
        self.__balance = balance  # Private attribute

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount

    def get_balance(self):
        return self.__balance

account = Account(1000)
print(account.get_balance())  # Output: 1000
```

2.2 Abstraction

Abstraction hides complexity by providing a simple interface. Only the essential features are presented, and complex details are hidden from the user.

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

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

    def area(self):
        return 3.14 * self.radius ** 2

circle = Circle(5)
print(circle.area())  # Output: 78.5
```

2.3 Inheritance

Inheritance allows a class (child class) to inherit attributes and methods from another class (parent class), enabling code reuse and extending functionality.

```
class Animal:
    def speak(self):
        return "Some sound"

class Dog(Animal):  # Dog inherits from Animal
    def speak(self):
        return "Woof!"

dog = Dog()
print(dog.speak())  # Output: Woof!
```

2.4 Polymorphism

Polymorphism allows different classes to be used interchangeably by providing a common interface. It lets methods perform different functions based on the object calling them.

```
class Cat:
    def speak(self):
        return "Meow!"

animals = [Dog(), Cat()]

for animal in animals:
    print(animal.speak())
```

---

3. Advanced Concepts in OOP

3.1 Method Overriding

When a child class provides a specific implementation of a method in the parent class, it's called method overriding. This enables customized behavior in child classes.

```python
class Animal:
    def speak(self):
        return "Some sound"

class Dog(Animal):
    def speak(self):
        return "Woof!"

dog = Dog()
print(dog.speak())  # Output: Woof!
```

3.2 Method Overloading (Python's Approach)

Python doesn't support traditional method overloading. Instead, you can use default arguments or `*args` and `**kwargs` to handle multiple argument types or numbers.

```python
class MathOperations:
    def add(self, a, b, c=0):
        return a + b + c

math_op = MathOperations()
print(math_op.add(2, 3))  # Output: 5
print(math_op.add(2, 3, 4))  # Output: 9
```

3.3 Operator Overloading

Python allows operators to be overloaded, meaning you can define custom behavior for operators with objects of custom classes.

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

    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)

v1 = Vector(1, 2)
v2 = Vector(3, 4)
result = v1 + v2  # Uses __add__ method
print(result.x, result.y)  # Output: 4 6
```

3.4 Multiple Inheritance

Python supports multiple inheritance, where a class can inherit from more than one class.

```
class Flyable:
    def fly(self):
        return "Flying"

class Swimmable:
    def swim(self):
        return "Swimming"

class Duck(Flyable, Swimmable):
    pass

duck = Duck()
print(duck.fly())  # Output: Flying
print(duck.swim())  # Output: Swimming
```

3.5 Composition

Composition is a design principle where a class is made up of other classes. This helps avoid the pitfalls of multiple inheritance.

```
class Engine:
    def start(self):
        return "Engine started"

class Car:
    def __init__(self):
        self.engine = Engine()  # Composition

    def start_engine(self):
        return self.engine.start()
```

```python
car = Car()
print(car.start_engine())  # Output: Engine started
```

---

4. Access Modifiers in Python

Python does not have strict access modifiers (`public`, `protected`, `private`) like other languages. Instead: - **Public**: Accessible anywhere. - **Protected (_attribute)**: A convention indicating that it should not be accessed outside the class or its subclasses. -** Private (__attribute)**: Internally renamed to prevent accidental access but still accessible using name mangling.

---

5. Best Practices in OOP

1. **Use Class and Instance Attributes Appropriately**: Define attributes at the class level if they're shared across all instances; otherwise, set them in `__init__`.
2. **Minimize Use of Public Attributes**: Use private/protected attributes for internal states.
3. **Avoid Deep Inheritance Chains**: Use composition over inheritance to keep the code simple.
4. **Leverage Polymorphism**: Write code that works with any subclass, enhancing flexibility.
5. **Document Class and Methods**: Use docstrings to explain what the class and methods do, which makes the code easier to understand.

Summary

OOP in Python provides a powerful way to structure your code, using classes, objects, inheritance, polymorphism, encapsulation, and abstraction. Understanding these concepts and how to apply them effectively can help create efficient, reusable, and scalable code.

```
### A class is a blue print for creating objects. Attributes,methods
class Car:
    pass

audi=Car()
bmw=Car()

print(type(audi))
```

```
print(audi)
print(bmw)
```

```
audi.windows=4

print(audi.windows)
```

```
dir(audi)
```

```
### Instance Variable and Methods
class Dog:
    ## constructor
    def __init__(self,name,age):
        self.name=name
        self.age=age

## create objects
dog1=Dog("Buddy",3)
print(dog1)
print(dog1.name)
print(dog1.age)
```

```
dog2=Dog("Lucy",4)
print(dog2.name)
print(dog2.age)
```

```
## Define a class with instance methods
class Dog:
    def __init__(self,name,age):
        self.name=name
        self.age=age
```

```python
    def bark(self):
        print(f"{self.name} says woof")


dog1=Dog("Buddy",3)
dog1.bark()
dog2=Dog("Lucy",4)
dog2.bark()
```

```python
### Modeling a Bank Account

## Define a class for bank account
class BankAccount:
    def __init__(self,owner,balance=0):
        self.owner=owner
        self.balance=balance

    def deposit(self,amount):
        self.balance+=amount
        print(f"{amount} is deposited. New balance is {self.balance}")

    def withdraw(self,amount):
        if amount>self.balance:
            print("Insufficient funds!")
        else:
            self.balance-=amount
            print(f"{amount} is withdrawn. New Balance is {self.balance}")

    def get_balance(self):
        return self.balance

## create an account

account=BankAccount("Krish",5000)
print(account.balance)
```

```python
## Call isntance methods
account.deposit(100)
account.withdraw(300)
```

```
print(account.get_balance())
```

Inheritance In Python

Inheritance is a fundamental concept in Object-Oriented Programming (OOP) that allows a class to inherit attributes and methods from another class. This lesson covers single inheritance and multiple inheritance, demonstrating how to create and use them in Python.

```
## Inheritance (Single Inheritance)
## Parent class
class Car:
    def __init__(self,windows,doors,enginetype):
        self.windows=windows
        self.doors=doors
        self.enginetype=enginetype

    def drive(self):
        print(f"The person will drive the {self.enginetype} car ")
```

```
car1=Car(4,5,"petrol")
car1.drive()
```

```
class Tesla(Car):
    def __init__(self,windows,doors,enginetype,is_selfdriving):
        super().__init__(windows,doors,enginetype)
        self.is_selfdriving=is_selfdriving

    def selfdriving(self):
        print(f"Tesla supports self driving : {self.is_selfdriving}")
```

```
tesla1=Tesla(4,5,"electric",True)
tesla1.selfdriving()
```

```
tesla1.drive()
```

```
### Multiple Inheritance
## When a class inherits from more than one base class.
## Base class 1
```

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

    def speak(self):
        print("Subclass must implement this method")

## Base class 2
class Pet:
    def __init__(self, owner):
        self.owner = owner


##Derived class
class Dog(Animal,Pet):
    def __init__(self,name,owner):
        Animal.__init__(self,name)
        Pet.__init__(self,owner)

    def speak(self):
        return f"{self.name} say woof"


## Create an object
dog=Dog("Buddy","Raja")
print(dog.speak())
print(f"Owner:{dog.owner}")
```

Conclusion

Inheritance is a powerful feature in OOP that allows for code reuse and the creation of a more logical class structure. Single inheritance involves one base class, while multiple inheritance involves more than one base class. Understanding how to implement and use inheritance in Python will enable you to design more efficient and maintainable object-oriented programs.

Polymorphism

Polymorphism is a core concept in Object-Oriented Programming (OOP) that allows objects of different classes to be treated as objects of a common superclass. It provides a way to perform a single action in different forms. Polymorphism is typically achieved through method overriding and interfaces

Method Overriding

Method overriding allows a child class to provide a specific implementation of a method that is already defined in its parent class.

```python
## Base Class
class Animal:
    def speak(self):
        return "Sound of the animal"

## Derived Class 1
class Dog(Animal):
    def speak(self):
        return "Woof!"

## Derived class
class Cat(Animal):
    def speak(self):
        return "Meow!"

## Function that demonstrates polymorphism
def animal_speak(animal):
    print(animal.speak())

dog=Dog()
cat=Cat()
print(dog.speak())
print(cat.speak())
animal_speak(dog)
```

```python
### Polymorphissm with Functions and MEthods
## base class
class Shape:
    def area(self):
        return "The area of the figure"

## Derived class 1
class Rectangle(Shape):
    def __init__(self,width,height):
        self.width=width
        self.height=height
    def area(self):
        return self.width * self.height

##Derived class 2

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

    def area(self):
        return 3.14*self.radius *self.radius

## Fucntion that demonstrates polymorphism

def print_area(shape):
    print(f"the area is {shape.area()}")


rectangle=Rectangle(4,5)
circle=Circle(3)

print_area(rectangle)
print_area(circle)
```

Polymorphism with Abstract Base Classes

Abstract Base Classes (ABCs) are used to define common methods for a group of related objects. They can enforce that derived classes implement particular methods, promoting consistency across different implementations.

Consider a scenario where you are building a payment processing system. You might have different methods of payment, like credit card, PayPal, or cryptocurrencies. An abstract class can define a template that all these payment methods must follow:

```
from abc import ABC, abstractmethod

class Payment(ABC):

    @abstractmethod
    def authorize(self, amount):
        pass
    @abstractmethod
    def capture(self):
        pass

    @abstractmethod
    def refund(self, amount):
        pass

class CreditCardPayment(Payment):

    def authorize(self, amount):
        print(f"Authorizing {amount} on credit card.")

    def capture(self):
        print("Capturing funds from the credit card.")

    def refund(self, amount):
        print(f"Refunding {amount} to the credit card.")

class PayPalPayment(Payment):

    def authorize(self, amount):
        print(f"Authorizing {amount} with PayPal.")

    def capture(self):
        print("Capturing funds from PayPal account.")

    def refund(self, amount):
        print(f"Refunding {amount} to the PayPal account.")

# Usage:
payment = CreditCardPayment()
payment.authorize(100)
payment.capture()
payment.refund(50)
```

```
from abc import ABC,abstractmethod

## Define an abstract class
class Vehicle(ABC):
    @abstractmethod
    def start_engine(self):
        pass

## Derived class 1
class Car(Vehicle):
    def start_engine(self):
        return "Car enginer started"

## Derived class 2
class Motorcycle(Vehicle):
    def start_engine(self):
        return "Motorcycle enginer started"

# Function that demonstrates polymorphism
def start_vehicle(vehicle):
    print(vehicle.start_engine())

## create objects of cAr and Motorcycle

car = Car()
motorcycle = Motorcycle()

start_vehicle(car)
```

```
from abc import ABC, abstractmethod

# Define an abstract class by inheriting from ABC
class Animal(ABC):

    # Define an abstract method using the @abstractmethod decorator
    @abstractmethod
    def make_sound(self):
        pass

    # A concrete method can be defined with an implementation
    def move(self):
        print("Moving...")

# Derived class from Animal that implements the abstract method
class Dog(Animal):

    def make_sound(self):
        print("Woof!")
```

```
# Another derived class from Animal
class Cat(Animal):

    def make_sound(self):
        print("Meow!")

# Attempting to instantiate an abstract class directly will raise an error
# animal = Animal()  # This will raise a TypeError

# Instantiate the concrete classes
dog = Dog()
cat = Cat()

# Call the implemented abstract methods
dog.make_sound()  # Outputs: Woof!
cat.make_sound()  # Outputs: Meow!

# Call the inherited concrete method
dog.move()  # Outputs: Moving...
cat.move()  # Outputs: Moving...
```

Encapsulation is a fundamental concept in object-oriented programming (OOP), and it focuses on bundling data and methods that operate on that data within a single unit, such as a class. In Python, encapsulation is commonly implemented through access modifiers, like making attributes private or protected to control access to an object's data.

Key Goals of Encapsulation

1. **Control Access to Data**: Encapsulation helps restrict direct access to certain attributes and methods, so external code can't modify them directly.
2. **Maintain Data Integrity**: It ensures data is modified only through controlled methods, preventing unwanted changes.
3. **Reduce Complexity**: By hiding the inner workings, encapsulation makes it easier to manage and understand code.

Encapsulation in Python with Examples

In Python, we use conventions and special syntax to control access to attributes. Here's a breakdown of common access levels and how to implement encapsulation.

1. Public Attributes and Methods

Public attributes are accessible from outside the class, and we define them without any leading underscores.

```python
class Car:
    def __init__(self, brand, model):
        self.brand = brand  # Public attribute
        self.model = model  # Public attribute

    def start_engine(self):  # Public method
        print(f"{self.brand} {self.model}'s engine started.")

# Example usage:
my_car = Car("Toyota", "Corolla")
print(my_car.brand)  # Accessible
my_car.start_engine()  # Accessible
```

2. Protected Attributes and Methods

Protected members are marked with a single underscore (_). This indicates that they are intended for internal use within the class and its subclasses, though they are still accessible outside.

```python
class Car:
    def __init__(self, brand, model, fuel_capacity):
        self._brand = brand       # Protected attribute
        self._model = model       # Protected attribute
        self._fuel_capacity = fuel_capacity  # Protected attribute

    def _check_fuel_level(self):  # Protected method
        return f"Fuel level checked for {self._brand} {self._model}."

# Example usage:
my_car = Car("Honda", "Civic", 50)
print(my_car._brand)  # Although discouraged, it’s accessible
print(my_car._check_fuel_level())  # Although discouraged, it’s accessible
```

3. Private Attributes and Methods

Private members use a double underscore (__) as a prefix, making them accessible only within the class itself. Python name-mangles these variables, so accessing them directly from outside the class is more challenging.

```python
class BankAccount:
    def __init__(self, account_holder, balance):
        self.account_holder = account_holder
        self.__balance = balance  # Private attribute

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
            print(f"Deposited {amount}. New balance: {self.__balance}")
        else:
            print("Deposit amount must be positive.")

    def get_balance(self):
        return self.__balance

# Example usage:
account = BankAccount("Alice", 1000)
account.deposit(500)  # Allowed
print(account.get_balance())  # Allowed, since get_balance is public
# print(account.__balance)  # Would raise an AttributeError due to encapsulation
```

If you attempt to access `__balance` directly, you'll get an error because it's been name-mangled by Python. However, it's technically possible to access it as `_BankAccount__balance`, but this is discouraged because it breaks encapsulation principles.

Example: Encapsulation with Setter and Getter Methods

Using getter and setter methods is a common practice in Python to control access to private attributes, ensuring data integrity and validation.

```python
class Student:
    def __init__(self, name, age):
        self.__name = name       # Private attribute
        self.__age = age         # Private attribute

    # Getter method for age
    def get_age(self):
        return self.__age

    # Setter method for age
    def set_age(self, age):
        if age > 0:
            self.__age = age
        else:
            print("Invalid age. Age must be positive.")

    # Getter for name
    def get_name(self):
        return self.__name

# Example usage:
student = Student("John", 20)
print(student.get_name())  # Accessing name via getter
print(student.get_age())   # Accessing age via getter
student.set_age(25)        # Setting a new age
print(student.get_age())   # Checking updated age
student.set_age(-5)        # Trying to set an invalid age
```

Here, direct access to `__age` and `__name` is restricted, so they can only be accessed or modified through the getter (`get_age`, `get_name`) and setter (`set_age`) methods. This maintains control over how `__age` and `__name` are modified.

Advantages of Encapsulation

- **Data Security**: Sensitive data is not exposed directly.
- **Controlled Access**: Through getter and setter methods, you can implement checks or validation, ensuring data integrity.
- **Improved Code Maintenance**: Since internal implementation details are hidden, changes inside the class won't affect code outside the class.

Example: Encapsulation in a Real-world Context

Let's consider a more practical example, a `Library` class where books can be borrowed and returned.

```
class Library:
    def __init__(self, name):
        self.__name = name
        self.__books = {}  # Private dictionary to hold books and availability

    def add_book(self, book):
        self.__books[book] = True  # True means the book is available
    def borrow_book(self, book):
        if self.__books.get(book):
            self.__books[book] = False
            print(f"You've borrowed '{book}' from {self.__name}.")
        else:
            print(f"'{book}' is currently unavailable.")

    def return_book(self, book):
        if book in self.__books:
            self.__books[book] = True
            print(f"Thank you for returning '{book}' to {self.__name}.")
        else:
            print(f"'{book}' does not belong to {self.__name}.")

# Example usage:
my_library = Library("City Library")
my_library.add_book("Python Programming")
my_library.borrow_book("Python Programming")  # Successfully borrows the book
my_library.borrow_book("Python Programming")  # Shows the book is unavailable
my_library.return_book("Python Programming")  # Successfully returns the book
```

Here, encapsulation keeps the book availability data private (`__books`), meaning that users can only interact with the books through controlled methods like `borrow_book` and `return_book`. This protects the internal structure and enforces a set process for borrowing and returning books.

This approach makes the code more maintainable and secure by not allowing direct changes to `__books`.

Encapsulation And Abstraction

Encapsulation and abstraction are two fundamental principles of Object-Oriented Programming (OOP) that help in designing robust, maintainable, and reusable code. Encapsulation involves bundling data and methods that operate on the data within a single unit, while abstraction involves hiding complex implementation details and exposing only the necessary features.

Encapsulation

Encapsulation is the concept of wrapping data (variables) and methods (functions) together as a single unit. It restricts direct access to some of the object's components, which is a means of preventing accidental interference and misuse of the data.

```
### Encapsulation  with Getter and Setter MEthods
### Public,protected,private variables or access modifiers

class Person:
    def __init__(self,name,age):
        self.name=name    ## public variables
        self.age=age      ## public variables

def get_name(person):
    return person.name

person=Person("Mukesh",27)
get_name(person)
```

```
dir(person)
```

```
class Person:
    def __init__(self,name,age,gender):
        self._name=name    ## protected variables
        self._age=age      ## protected variables
        self.gender=gender

class Employee(Person):
    def __init__(self,name,age,gender):
        super().__init__(name,age,gender)


employee=Employee("Mukesh",24,"Male")
print(employee._name)
```

```
## Encapsulation With Getter And Setter
class Person:
    def __init__(self,name,age):
        self.__name=name  ## Private access modifier or variable
        self.__age=age ## Private variable

    ## getter method for name
    def get_name(self):
        return self.__name

    ## setter method for name
    def set_name(self,name):
        self.__name=name

    # Getter method for age
    def get_age(self):
        return self.__age

    # Setter method for age
    def set_age(self, age):
        if age > 0:
            self.__age = age
        else:
            print("Age cannot be negative.")


person=Person("Mukesh",34)

## Access and modify private variables using getter and setter

print(person.get_name())
print(person.get_age())

person.set_age(35)
print(person.get_age())

person.set_age(-5)
```

Abstraction

Abstraction is the concept of hiding the complex implementation details and showing only the necessary features of an object. This helps in reducing programming complexity and effort.

```
from abc import ABC,abstractmethod

## Abstract base cclass
class Vehicle(ABC):
    def drive(self):
        print("The vehicle is used for driving")

    @abstractmethod
    def start_engine(self):
        pass

class Car(Vehicle):
    def start_engine(self):
        print("Car enginer started")

def operate_vehicle(vehicle):
    vehicle.start_engine()
    vehicle.drive()

car=Car()
operate_vehicle(car)
```

Magic Methods

Magic methods in Python, also known as dunder methods (double underscore methods), are special methods that start and end with double underscores. These methods enable you to define the behavior of objects for built-in operations, such as arithmetic operations, comparisons, and more.

Magic Methods

Magic methods are predefined methods in Python that you can override to change the behavior of your objects. Some common magic methods include:

```
# '''
# __init__': Initializes a new instance of a class.
# __str__: Returns a string representation of an object.
# __repr__: Returns an official string representation of an object.
# __len__: Returns the length of an object.
# __getitem__: Gets an item from a container.
# __setitem__: Sets an item in a container.
# '''
```

```
class Person:
    pass

person=Person()
dir(person)
```

```
## Basics MEthods
class Person:
    def __init__(self,name,age):
        self.name=name
        self.age=age
person=Person("KRish",34)
print(person)
```

```
## Basics MEthods
class Person:
    def __init__(self,name,age):
        self.name=name
        self.age=age

    def __str__(self):
        return f"{self.name},{self.age} years old"

    def __repr__(self):
        return f"Person(name={self.name},age={self.age})"

person=Person("KRish",34)
print(person)
print(repr(person))
```

Operator Overloading

Operator overloading allows you to define the behavior of operators (+, -, *, etc.) for custom objects. You achieve this by overriding specific magic methods in your class.‘

```
#### Common Operator Overloading Magic Methods

# __add__(self, other): Adds two objects using the + operator.
# __sub__(self, other): Subtracts two objects using the - operator.
# __mul__(self, other): Multiplies two objects using the * operator.
# __truediv__(self, other): Divides two objects using the / operator.
# __eq__(self, other): Checks if two objects are equal using the == operator.
# __lt__(self, other): Checks if one object is less than another.
# __gt__(self, other): Checks if one object is greater than another.
```

```
### MAthematical operation for vectors
class Vector:
    def __init__(self,x,y):
        self.x=x
        self.y=y

    def __add__(self,other):
        return Vector(self.x+other.x,self.y+other.y)

    def __sub__(self, other):
        return Vector(self.x - other.x, self.y - other.y)

    def __mul__(self, other):
        return Vector(self.x * other, self.y * other)

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

    def __repr__(self):
        return f"Vector({self.x}, {self.y})"

## create objectss of the Vector Class
v1=Vector(2,3)
v2=Vector(4,5)

print(v1+v2)
print(v1-v2)
print(v1*3)
```

```
### Overloading Operators for Complex Numbers

class ComplexNumber:
    def __init__(self, real, imag):
        self.real = real
        self.imag = imag

    def __add__(self, other):
        return ComplexNumber(self.real + other.real, self.imag + other.imag)

    def __sub__(self, other):
        return ComplexNumber(self.real - other.real, self.imag - other.imag)

    def __mul__(self, other):
        real_part = self.real * other.real - self.imag * other.imag
        imag_part = self.real * other.imag + self.imag * other.real
        return ComplexNumber(real_part, imag_part)

    def __truediv__(self, other):
        denom = other.real**2 + other.imag**2
        real_part = (self.real * other.real + self.imag * other.imag) / denom
        imag_part = (self.imag * other.real - self.real * other.imag) / denom
        return ComplexNumber(real_part, imag_part)

    def __eq__(self, other):
        return self.real == other.real and self.imag == other.imag

    def __repr__(self):
        return f"{self.real} + {self.imag}i"

# Create objects of the ComplexNumber class
c1 = ComplexNumber(2, 3)
c2 = ComplexNumber(1, 4)

# Use overloaded operators
print(c1 + c2)  # Output: 3 + 7i
print(c1 - c2)  # Output: 1 - 1i
print(c1 * c2)  # Output: -10 + 11i
print(c1 / c2)  # Output: 0.8235294117647058 - 0.29411764705882354i
print(c1 == c2) # Output: False
```

```
### Custom exception (Raise and Throw an exception)
class Error(Exception):
    pass

class dobException(Error):
    pass
```

```
year=1997
age=2024-year

try:
    if age<=30 and age>=20:
        print("The age is valid so you can apply for the exam")
    else:
        raise dobException
except dobException:
    print("Sorry,your age should be greater than 20 or less than 30")
```

1. Basic Concept of Decorators

A decorator is essentially a function that takes another function (or method) as an argument and extends its behavior without modifying its structure. This is useful for adding functionality to code in a reusable way.

Certainly! Here are the examples with the expected output added as comments to help illustrate what happens at each stage.

---

2. Function Wrapping Example with Output

```
def my_decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper

# Using the decorator on a function
def say_hello():
    print("Hello!")

say_hello = my_decorator(say_hello)
say_hello()

# Output:
# Something is happening before the function is called.
# Hello!
# Something is happening after the function is called.
```

---

3. Syntactic Sugar Example with Output

```python
@my_decorator
def say_hello():
    print("Hello!")

say_hello()

# Output:
# Something is happening before the function is called.
# Hello!
# Something is happening after the function is called.
```

---

4. Decorators with Arguments Example with Output

```python
def repeat(num_times):
    def decorator_repeat(func):
        def wrapper(*args, **kwargs):
            for _ in range(num_times):
                func(*args, **kwargs)
        return wrapper
    return decorator_repeat

@repeat(num_times=3)
def greet(name):
    print(f"Hello {name}!")

greet("Alice")

# Output:
# Hello Alice!
# Hello Alice!
# Hello Alice!
```

---

5. Timing Decorator Example with Output

```
import time

def timer(func):
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        print(f"{func.__name__} took {end_time-start_time:.4f} sec to exec")
        return result
    return wrapper

@timer
def some_function():
    time.sleep(1)

some_function()

# Output (time will vary slightly):
# some_function took 1.0002 seconds to execute
```

---

6. Decorators on Methods Example with Output

```
def uppercase(func):
    def wrapper(*args, **kwargs):
        original_result = func(*args, **kwargs)
        return original_result.upper()
    return wrapper

class Greeter:
    @uppercase
    def greet(self):
        return "hello, world"

g = Greeter()
print(g.greet())
```

```
# Output:
# HELLO, WORLD
```

---

7. Using `functools.wraps` Example with Output

```python
from functools import wraps

def my_decorator(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print("Before the function")
        result = func(*args, **kwargs)
        print("After the function")
        return result
    return wrapper

@my_decorator
def say_hello():
    """Greet the user"""
    print("Hello!")

say_hello()
print(say_hello.__name__)  # Outputs: say_hello, not wrapper
print(say_hello.__doc__)   # Outputs: "Greet the user"

# Output:
# Before the function
# Hello!
# After the function
# say_hello
# Greet the user
```

---

8. Chaining Decorators Example with Output

```python
def bold(func):
    def wrapper():
        return f"<b>{func()}</b>"
    return wrapper

def italic(func):
    def wrapper():
        return f"<i>{func()}</i>"
    return wrapper

@bold
@italic
def hello():
    return "Hello!"

print(hello())

# Output:
# <b><i>Hello!</i></b>
```

---

10. Practical Example: Caching with `functools.lru_cache`

```python
from functools import lru_cache

@lru_cache(maxsize=32)
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

print(fibonacci(10))  # Cached results improve performance
print(fibonacci.cache_info())  # Shows cache statistics

# Output:
# 55
# CacheInfo(hits=9, misses=11, maxsize=32, currsize=11)
```

Generators in Python are a special type of iterable that allow you to lazily evaluate sequences of values, meaning they generate items one at a time only when needed, instead of storing them all in memory at once. They are extremely useful for handling large datasets, streams of data, or any situation where you want efficient memory usage and deferred computation. Here's everything you need to know about generators in Python:

1. What is a Generator?

A generator in Python is a function that returns an iterator, which we can iterate over (one value at a time). Unlike a normal function, which returns a single value and then exits, a generator can yield multiple values, pausing its execution and resuming where it left off each time.

2. Creating a Generator: The `yield` Keyword

A generator function uses the `yield` keyword to return a value temporarily, unlike `return`, which would end the function execution entirely. Every time `yield` is encountered, the generator "yields" control back to the caller, saving its state for later resumption.

```
def my_generator():
    yield 1
    yield 2
    yield 3

# Usage
for value in my_generator():
    print(value)
```

This would output: `1  2  3`

3. How Generators Work Under the Hood

Each time the generator's `__next__()` method is called (either by the `next()` function or by a `for` loop), it resumes from where it last yielded, runs to the next `yield` statement, and pauses again. When there are no more `yield` statements, it raises a `StopIteration` exception, signaling the end of the iteration.

4. Benefits of Generators

- **Memory Efficiency:** Generators don't store all values in memory; they yield items one at a time. This is useful for large datasets.
- **Lazy Evaluation:** Values are produced only when requested, which can improve performance when dealing with long or infinite sequences.
- **Composability:** Generators can be chained or combined easily to build complex data pipelines.

5. Generator Expressions

Similar to list comprehensions, generator expressions allow you to create a generator in a concise way. Generator expressions are surrounded by parentheses `()` instead of square brackets `[]`.

```
squares = (x * x for x in range(10))
```

- Unlike list comprehensions, generator expressions do not generate all items at once, but yield them one at a time.

6. Example: Using Generators for Large Data Processing

If we have a large file and only need to process one line at a time, a generator can be very efficient:

```
def read_large_file(file_path):
    with open(file_path) as file:
        for line in file:
            yield line

# Usage
for line in read_large_file("big_data.txt"):
    process(line)  # process each line as needed
```

7. Infinite Generators

Generators can be useful to create an infinite sequence of values. Just be cautious with them to avoid infinite loops in code that consumes these generators.

```
def infinite_counter():
    num = 0
    while True:
        yield num
        num += 1
```

```
# Usage with break condition
for i in infinite_counter():
    if i > 5:
        break
    print(i)
```

8. Sending Values to Generators

Generators can receive data from the outside using the `.send()` method. This allows a generator to act like a coroutine, where values can be sent into it while it's running.

```
def generator_with_send():
    while True:
        received = yield
        print(f'Received: {received}')

gen = generator_with_send()
next(gen)        # Start the generator
gen.send("Hello")
gen.send("World")
```

9. Closing a Generator

Generators can be manually closed with the `.close()` method. This raises a `GeneratorExit` exception inside the generator, which can be caught if you want to perform any cleanup.

```
def my_generator():
    try:
        yield 1
        yield 2
    except GeneratorExit:
        print("Generator closed!")

gen = my_generator()
```

```
next(gen)
gen.close()  # Triggers GeneratorExit inside the generator
```

10. Generator Lifecycle Summary

- **Creation:** A generator is created by calling a generator function or using a generator expression.
- **Start/Resume:** Begins execution on the first `next()` call.
- **Yielding Values:** Each `yield` pauses the generator, sending a value back to the caller.
- **Ending:** It stops when the function exits, or if `StopIteration` or `GeneratorExit` exceptions are raised.

Practical Examples

1. **Fibonacci Sequence:**

   ```
   def fibonacci():
       a, b = 0, 1
       while True:
           yield a
           a, b = b, a + b
   ```

2. **Chunking Data from a List:**

   ```
   def chunk_data(data, chunk_size):
       for i in range(0, len(data), chunk_size):
           yield data[i:i + chunk_size]
   ```

Generators are a core tool in Python for handling large data, creating efficient pipelines, and supporting memory-efficient programs. Would you like examples of advanced uses, or to dive into specific patterns?

Iterators in Python are a powerful way to access elements in a collection (like lists, tuples, or dictionaries) one at a time. They allow lazy evaluation, which means that elements are only generated when requested, making them memory-efficient and suitable for large data sets. Here's an in-depth look at how iterators work in Python:

1. Basic Concept of Iterators

- **Iterator**: An object that represents a stream of data. It implements the iterator protocol, which includes methods like `__iter__()` and `__next__()`.

- **Iterable**: An object that can return an iterator, like lists, tuples, strings, and dictionaries. Any object with an `__iter__()` method is an iterable.

2. How Iterators Work

- To use an iterator, Python first calls `__iter__()` on an iterable object to get an iterator.
- The iterator then uses its `__next__()` method to fetch the next element. When there are no more elements, it raises a `StopIteration` exception.

3. Creating an Iterator

- In Python, we usually use built-in iterators like lists and dictionaries. However, you can also create custom iterators by defining a class with `__iter__()` and `__next__()` methods.

Here's a custom iterator example:

```python
class Counter:
    def __init__(self, limit):
        self.limit = limit
        self.count = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.count < self.limit:
            self.count += 1
            return self.count
        else:
            raise StopIteration

counter = Counter(5)
for number in counter:
    print(number)
```

In this example, `Counter` is an iterator that counts up to the specified limit.

4. Using the `iter()` and `next()` Functions

- `iter()` can turn an iterable into an iterator.
- `next()` retrieves the next element from an iterator.

Example:

```python
numbers = [1, 2, 3]
iterator = iter(numbers)
print(next(iterator))  # Output: 1
print(next(iterator))  # Output: 2
print(next(iterator))  # Output: 3
```

5. Generators: Simplified Iterators

- Generators are functions that yield values one at a time, allowing you to create iterators without explicitly implementing `__iter__()` and `__next__()`.
- Use the `yield` keyword instead of `return`. Each call to `next()` resumes the function from where it left off.

Example:

```python
def countdown(n):
    while n > 0:
        yield n
        n -= 1

for num in countdown(5):
    print(num)
```

6. Infinite Iterators

- Python has infinite iterators in the `itertools` module (e.g., `itertools.count` and `itertools.cycle`).
- Be cautious with infinite iterators, as they don't end unless explicitly broken.

7. Common Use Cases for Iterators

- **Lazy Evaluation**: Ideal for processing large datasets where not all elements are needed at once.
- **Data Pipelines**: Used in data processing workflows, especially when paired with functions like `map()`, `filter()`, and `zip()`.
- **Efficient Resource Management**: Iterators only compute the value on demand, saving memory.

8. Advantages of Iterators

- **Memory Efficiency**: Since they don't require loading all elements at once, iterators are memory-friendly.
- **Pipeline Capability**: Work well with function chaining in a pipeline (e.g., applying transformations to data streams).

9. Limitations of Iterators

- **One-Time Use**: Once an iterator is exhausted, you need a new one to re-iterate.
- **No Random Access**: Unlike lists, you can't access elements by index.

```python
import threading
import time

def print_numbers():
    for i in range(5):
        print(f"Number: {i}")
        time.sleep(1)

def print_letters():
    for letter in 'ABCDE':
        print(f"Letter: {letter}")
        time.sleep(1)

# Create threads
t1 = threading.Thread(target=print_numbers)
t2 = threading.Thread(target=print_letters)

# Start threads
t1.start()
t2.start()
```

```
# Wait for threads to finish
t1.join()
t2.join()
```

Multithreading Example

```
import threading
import time

def print_numbers():
    for i in range(5):
        print(f"Number: {i}")
        time.sleep(1)

def print_letters():
    for letter in 'ABCDE':
        print(f"Letter: {letter}")
        time.sleep(1)

# Create threads
t1 = threading.Thread(target=print_numbers)
t2 = threading.Thread(target=print_letters)

# Start threads
t1.start()
t2.start()

# Wait for threads to finish
t1.join()
t2.join()
```

Explanation:

- **Multithreading**: This code creates two threads, each running a separate function.
- **Functions**:
  - `print_numbers` prints numbers from 0 to 4, pausing for 1 second after each.
  - `print_letters` prints letters from 'A' to 'E', also pausing for 1 second after each.
- **Execution**:

    - `t1.start()` and `t2.start()` run both threads concurrently.
    - `t1.join()` and `t2.join()` ensure the main program waits for both threads to complete before exiting.

- **Output**: The threads run concurrently, so the output interleaves numbers and letters based on timing.

Multithreading with Locks Example

```python
import threading

counter = 0
lock = threading.Lock()

def increment_counter():
    global counter
    for _ in range(1000):
        with lock:
            counter += 1

threads = [threading.Thread(target=increment_counter) for _ in range(10)]

for thread in threads:
    thread.start()

for thread in threads:
    thread.join()

print(f"Final counter: {counter}")
```

Explanation:

- **Race Condition**: When multiple threads access and modify shared data (`counter`), conflicts can occur.
- **Lock Usage**:
    - A `lock` ensures that only one thread modifies the `counter` at a time.
    - `with lock:` creates a lock-protected code block.
- **Thread Creation**: Creates 10 threads, each running `increment_counter`.

- **Execution**: Each thread increments `counter` 1,000 times, and the `lock` prevents data corruption.
- **Expected Output**: `Final counter: 10000`, as each of the 10 threads increments `counter` 1,000 times.

Multiprocessing Example

```python
from multiprocessing import Process
import os

def print_process_info():
    print(f"Process ID: {os.getpid()}")

if __name__ == "__main__":
    processes = [Process(target=print_process_info) for _ in range(3)]

    for process in processes:
        process.start()

    for process in processes:
        process.join()
```

Explanation:

- **Multiprocessing**: This code demonstrates running separate processes (not threads).
- **Process Creation**:
  - The `print_process_info` function prints the process ID of each created process.
  - `Process(target=print_process_info)` creates a new process that runs the function.
- **Execution**:
  - Each process runs independently with its own memory space.
  - `process.start()` launches each process.
  - `process.join()` waits for each process to finish.
- **Output**: The output shows distinct process IDs because each process is independent.

Multiprocessing with Queue Example

```python
from multiprocessing import Process, Queue

def worker(q):
    q.put("Data from worker")

if __name__ == "__main__":
    q = Queue()
    p = Process(target=worker, args=(q,))
    p.start()
    p.join()
    print(q.get())
```

Explanation:

- **Queue for Inter-Process Communication (IPC)**:
  - A `Queue` allows communication between processes.
- **Process Execution**:
  - The `worker` function puts a message into the queue.
  - The main process reads the message from the queue after the worker process completes.
- **Output**: The main process retrieves "Data from worker" from the queue and prints it.

Asyncio Example

```python
import asyncio

async def task(name, delay):
    await asyncio.sleep(delay)
    print(f"Task {name} completed")

async def main():
    await asyncio.gather(task("A", 2), task("B", 1))

asyncio.run(main())
```

Explanation:

- **Asyncio**: Provides asynchronous programming using `async` and `await`.
- **Tasks and Delays**:
  - `task` function simulates work with `asyncio.sleep(delay)`.
  - `asyncio.gather` runs `task("A", 2)` and `task("B", 1)` concurrently.
- **Execution**:
  - `await asyncio.sleep(delay)` pauses without blocking other tasks.
  - `Task B` finishes before `Task A` due to the shorter delay.
- **Output**: Shows "Task B completed" first, then "Task A completed" as per delay timings.

Each example illustrates a different concurrency approach in Python, with multithreading, multiprocessing, and asyncio suited to different scenarios: multithreading for lightweight tasks, multiprocessing for CPU-bound work, and asyncio for I/O-bound tasks with minimal blocking.

What is the Python Standard Library?

The standard library in Python contains pre-written code for performing various tasks, such as handling files, working with data, interacting with the operating system, performing mathematical operations, and much more. Since it's bundled with Python, it's cross-platform, meaning you don't need to install these modules separately—they're ready to use once you install Python.

Why Use the Standard Library?

1. **Reliability and Performance**: Since it's part of Python's core, it's optimized for performance and thoroughly tested.
2. **Cross-Platform Compatibility**: Modules in the standard library are designed to work on multiple platforms.
3. **Convenience and Productivity**: By using built-in modules, you don't need to write code from scratch for common tasks.

Key Modules in the Python Standard Library

Here's a summary of some commonly used modules along with examples of how to use them:

1. **Math and Statistics: `math`, `cmath`, `random`, `statistics`**

   - **`math`**: Provides mathematical functions like square roots, trigonometric functions, logarithmic calculations, etc.
   - **`random`**: For generating random numbers, shuffling data, etc.
   - **`statistics`**: Basic statistical functions such as mean, median, and standard deviation.
   - **Example**:

```python
import math
print(math.sqrt(16))          # Output: 4.0
import random
print(random.randint(1, 10))  # Output: Random integer between 1 and 10
```

2. **String Manipulation: `re`, `string`, `textwrap`**

   - **`re`**: Handles regular expressions, allowing pattern matching in strings.
   - **`string`**: Contains constants and functions for string manipulation.
   - **Example**:

```python
import re
text = "Hello, welcome to Python!"
print(re.findall(r'\b\w{5}\b', text))  # Output: ['Hello']
```

3. **File and Directory Access: `os`, `shutil`, `pathlib`, `glob`**

   - **`os`**: Used for interacting with the operating system, file handling, and environment variables.
   - **`pathlib`**: Provides an object-oriented way of handling filesystem paths.
   - **Example**:

```python
import os
print(os.getcwd())  # Output: Current working directory
```

4. **Data Serialization: `json`, `csv`, `pickle`, `xml.etree.ElementTree`**

   - **`json`**: For encoding and decoding JSON.
   - **`csv`**: Makes it easy to read from and write to CSV files.
   - **Example**:

```python
import json
data = {'name': 'Alice', 'age': 30}
json_data = json.dumps(data)
print(json_data)  # Output: '{"name": "Alice", "age": 30}'
```

5. Date and Time: `datetime`, `time`, `calendar`

- **`datetime`**: Provides date and time manipulation capabilities.
- **Example**:

```python
from datetime import datetime
now = datetime.now()
print(now.strftime("%Y-%m-%d %H:%M:%S"))
# Output: '2024-11-09 15:35:29' (Example)
```

6. Networking: `socket`, `http`, `urllib`

- **`socket`**: Low-level networking interface.
- **`urllib`**: Fetching data across the web.
- **Example**:

```python
 import urllib.request
 with urllib.request.urlopen('http://example.com') as response:
     html = response.read()
```

7. Concurrent Programming: `threading`, `multiprocessing`, `asyncio`

- **`threading`**: For running tasks concurrently in multiple threads.
- **`multiprocessing`**: For parallel processing, which uses separate memory spaces for each process.
- **Example**:

```python
import threading
def print_numbers():
 for i in range(5):
     print(i)
thread = threading.Thread(target=print_numbers)
thread.start()
```

8. Debugging and Testing: `logging`, `unittest`, `doctest`

- **`logging`**: Helps in tracking events and issues in code execution.
- **`unittest`**: Framework for writing and running tests.
- **Example**:

```python
import logging
logging.basicConfig(level=logging.DEBUG)
logging.debug('This is a debug message')
```

9. System and Command-Line Interaction: `sys`, `argparse`, `subprocess`

- **`sys`**: Access system-specific parameters and functions.
- **`argparse`**: Easily parse command-line arguments.
- **Example**:

```python
import sys
print(sys.version)  # Output: Python version info
```

10. Data Collections and Itertools: `collections`, `itertools`, `functools`

- **`collections`**: Specialized container datatypes like named tuples, deque, Counter, etc.
- **`itertools`**: Tools for working with iterators.
- **Example**:

```python
from collections import Counter
print(Counter("apple"))  # Output: Counter({'a': 1, 'p': 2, 'l': 1, 'e': 1})
```

Tips for Using the Standard Library

1. **Use Documentation**: The official Python documentation (docs.python.org) is extensive and a great resource for understanding modules.
2. **Try Interactive Mode**: Experiment with the standard library modules in the Python interactive shell.
3. **Explore Hidden Gems**: Modules like `inspect` (for introspection), `pdb` (for debugging), and `concurrent.futures` (for parallel processing) can be very helpful.
4. **Know Module Limitations**: If a task goes beyond the library's capabilities (e.g., complex web scraping), look at external packages.

If you have a particular area you'd like to delve deeper into, such as web development, data processing, or system automation, I can provide more specific insights on those modules!

Let's dive deeper into some popular Python standard library modules, providing more practical examples and illustrating different use cases.

1. Math Module

The `math` module provides access to mathematical functions.

```
import math

# Trigonometric functions
print(math.sin(math.pi / 2))    # Output: 1.0

# Logarithmic functions
print(math.log(10))             # Natural log, Output: 2.302585092994046
print(math.log10(10))           # Log base 10, Output: 1.0

# Power and exponential
print(math.pow(2, 3))           # Output: 8.0 (2 raised to the power 3)
print(math.exp(2))              # Output: 7.389056098930649 (e^2)

# Rounding
print(math.ceil(2.3))           # Output: 3
print(math.floor(2.3))          # Output: 2
```

2. Random Module

The `random` module allows generating random numbers, useful in simulations, games, and sampling.

```
import random

# Random integer between 1 and 100
print(random.randint(1, 100))

# Random float between 0 and 1
print(random.random())

# Random choice from a list
```

```python
choices = ['apple', 'banana', 'cherry']
print(random.choice(choices))

# Shuffling a list
deck = ['Ace', 'King', 'Queen', 'Jack']
random.shuffle(deck)
print(deck)

# Random sample of 2 items from a list
print(random.sample(choices, 2))
```

3. String Manipulation with `re` and `string`

Regular expressions can be very useful for complex string operations.

```python
import re
import string

# Finding all words starting with "P" in a text
text = "Python is powerful and popular."
words = re.findall(r'\bP\w+', text)
print(words)  # Output: ['Python', 'powerful', 'popular']

# Checking if a string is a valid email
email = "test@example.com"
pattern = r'^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+$'
if re.match(pattern, email):
    print("Valid email")
else:
    print("Invalid email")

# Using string constants
print(string.ascii_lowercase)  # Output: 'abcdefghijklmnopqrstuvwxyz'
print(string.ascii_uppercase)  # Output: 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

4. File and Directory Access with `os`, `shutil`, and `pathlib`

Working with files and directories is essential in scripting and automation.

```
import os
from pathlib import Path

# Check if a directory exists, if not, create it
directory = Path("test_dir")
if not directory.exists():
    directory.mkdir()

# Listing files in the current directory
print(os.listdir('.'))

# Removing a directory (and its contents) with shutil
import shutil
shutil.rmtree('test_dir')

# Working with file paths using pathlib
file_path = Path("example.txt")
print(file_path.name)         # Output: 'example.txt'
print(file_path.stem)         # Output: 'example'
print(file_path.suffix)       # Output: '.txt'
```

5. Data Serialization with `json` and `csv`

Storing and loading data in formats like JSON and CSV is common in data processing.

```
import json
import csv

# JSON Example
data = {'name': 'Alice', 'age': 25, 'city': 'Wonderland'}
json_data = json.dumps(data)  # Serialize dictionary to JSON string
print(json_data)

# Writing JSON to a file
with open('data.json', 'w') as f:
    json.dump(data, f)

# Reading JSON from a file
with open('data.json', 'r') as f:
    loaded_data = json.load(f)
print(loaded_data)
```

```
# CSV Example
data = [
    ['name', 'age', 'city'],
    ['Alice', 25, 'Wonderland'],
    ['Bob', 30, 'Narnia']
]

# Writing to CSV
with open('data.csv', 'w', newline='') as f:
    writer = csv.writer(f)
    writer.writerows(data)

# Reading from CSV
with open('data.csv', 'r') as f:
    reader = csv.reader(f)
    for row in reader:
        print(row)
```

6. Datetime Module

Managing dates and times is crucial for logging, scheduling, and data analysis.

```
from datetime import datetime, timedelta

# Current date and time
now = datetime.now()
print("Now:", now)

# Formatting date
print("Formatted date:", now.strftime("%Y-%m-%d %H:%M:%S"))

# Parsing date from string
date_str = "2023-01-01 15:30:00"
parsed_date = datetime.strptime(date_str, "%Y-%m-%d %H:%M:%S")
print("Parsed date:", parsed_date)

# Date arithmetic
future_date = now + timedelta(days=5)
print("Future date:", future_date)

# Difference between dates
```

```python
difference = future_date - now
print("Days between:", difference.days)
```

7. Threading and Concurrency with `threading` and `asyncio`

Python offers modules for concurrent execution and async programming.

```python
import threading

# Using threads for concurrency
def print_numbers():
    for i in range(5):
        print(i)

thread1 = threading.Thread(target=print_numbers)
thread2 = threading.Thread(target=print_numbers)

thread1.start()
thread2.start()

# Using asyncio for async programming
import asyncio

async def async_task():
    print("Start task")
    await asyncio.sleep(1)
    print("End task")

# Run asyncio task
asyncio.run(async_task())
```

8. Debugging and Logging with `logging` and `unittest`

Logging is critical for production-ready applications.

```python
import logging

# Basic configuration for logging
logging.basicConfig(
    level=logging.DEBUG, format='%(asctime)s - %(levelname)s - %(message)s'
)

logging.debug('This is a debug message')
logging.info('This is an info message')
logging.warning('This is a warning message')
logging.error('This is an error message')
logging.critical('This is a critical message')

# Unit testing with unittest
import unittest

class TestMathFunctions(unittest.TestCase):
    def test_addition(self):
        self.assertEqual(1 + 1, 2)

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

9. System and Command-Line Interaction with `sys` and `argparse`

Useful for system-level programming and command-line interfaces.

```
import sys
import argparse

# Access command-line arguments
print("Arguments:", sys.argv)

# Using argparse for handling CLI arguments
parser = argparse.ArgumentParser(description="Simple CLI example")
parser.add_argument("name", help="Your name")
parser.add_argument("--age", type=int, help="Your age", default=0)

args = parser.parse_args()
print(f"Hello {args.name}, you are {args.age} years old.")
```

These examples should give you a strong understanding of the utility of Python's standard library. Each module has a lot more to offer, so let me know if you'd like to explore any specific one further!

The `os` and `pathlib` libraries in Python are powerful tools for interacting with the file system. They let you work with files, directories, and system paths effectively and offer extensive support for cross-platform compatibility.

1. `os` Module

The `os` module provides functions to interact with the operating system. It allows you to manipulate file paths, create or delete files and directories, check for files' existence, and more.

Commonly Used Functions in `os` Module

```
import os
```

- **Get Current Working Directory**:

  ```
  cwd = os.getcwd()
  print("Current Working Directory:", cwd)
  ```

- **Change Directory**:

  ```
  os.chdir('/path/to/directory')
  print("Changed Working Directory:", os.getcwd())
  ```

- **List Files and Directories**:

  ```
  # List files and directories in the current directory
  print(os.listdir('.'))
  ```

- **Create a Directory**:

  ```
  os.mkdir('new_folder')  # Creates a new directory
  os.makedirs('parent/child_folder')  # Creates nested directories
  ```

- **Remove a Directory**:

  ```
  os.rmdir('new_folder')  # Removes a single directory (must be empty)
  os.removedirs('parent/child_folder')  # Removes nested directories
  ```

- **Check if a Path Exists**:

  ```
  # Returns True if path exists, else False
  print(os.path.exists('some_file.txt'))
  ```

- **Join Paths**:

  ```
  path = os.path.join('folder', 'subfolder', 'file.txt')
  # Output: 'folder/subfolder/file.txt' on Linux, '
  # folder\\subfolder\\file.txt' on Windows
  print(path)
  ```

- **Get File Information**:

  ```
  file_info = os.stat('some_file.txt')
  print(file_info.st_size)    # File size in bytes
  print(file_info.st_mtime)   # Last modified time
  ```

- **Execute System Commands**:

  ```
  os.system('echo Hello World')  # Runs a system command
  ```

2. `pathlib` Module

The `pathlib` module provides an object-oriented approach to working with file paths and directories. It's part of Python 3's standard library and is cross-platform, making path handling simpler.

Working with `pathlib`

```
from pathlib import Path
```

- **Get the Current Working Directory**:

  ```
  cwd = Path.cwd()
  print("Current Working Directory:", cwd)
  ```

- **Create a Path Object**:

  ```
  path = Path('folder/subfolder/file.txt')
  print(path)  # Prints 'folder/subfolder/file.txt'
  ```

- **Join Paths**:

  ```
  new_path = Path('folder') / 'subfolder' / 'file.txt'
  print(new_path)  # Output: 'folder/subfolder/file.txt'
  ```

- **Check if Path Exists**:

```python
print(path.exists())       # True if path exists
print(path.is_file())      # True if path is a file
print(path.is_dir())       # True if path is a directory
```

- **Create Directories**:

```python
path = Path('new_folder')
# Creates directory, including parents if necessary
path.mkdir(parents=True, exist_ok=True)
```

- **Iterate Through Directory Contents**:

```python
for item in path.iterdir():
    print(item)  # Lists all files and directories in path
```

- **Read and Write Files**:

```python
# Writing to a file
file_path = Path('example.txt')
file_path.write_text("Hello, Pathlib!")  # Writes text to the file

# Reading from a file
content = file_path.read_text()
print(content)  # Output: 'Hello, Pathlib!'
```

- **Get File Properties**:

```python
file_path = Path('example.txt')
print(file_path.name)       # 'example.txt'
print(file_path.stem)       # 'example' (file name without extension)
print(file_path.suffix)     # '.txt' (file extension)
print(file_path.parent)     # Parent directory
```

- **Get File Size and Modification Time**:

```python
print(file_path.stat().st_size)  # File size in bytes
print(file_path.stat().st_mtime)  # Last modified time
```

Comparing `os` and `pathlib`

- **Path Joining**:
  - `os.path.join()` is platform-dependent and has to handle path separators manually, while `Path` objects with `/` in `pathlib` take care of platform differences.
- **File Operations**:

  - Both modules provide file operations, but `pathlib` is more intuitive and Pythonic with methods directly tied to `Path` objects, like `.write_text()` or `.read_text()`.

- **Cross-Platform**:
  - `pathlib` is often preferred in modern codebases for its clean, object-oriented syntax and automatic handling of OS differences.

Example: Moving a File with `os` and `pathlib`

Using `os`:

```
import os
import shutil

src = 'source_folder/file.txt'
dst = 'destination_folder/file.txt'

# Create destination folder if it doesn't exist
if not os.path.exists('destination_folder'):
    os.makedirs('destination_folder')

shutil.move(src, dst)
```

Using `pathlib`:

```
from pathlib import Path

src = Path('source_folder/file.txt')
dst = Path('destination_folder/file.txt')

# Create destination folder if it doesn't exist
dst.parent.mkdir(parents=True, exist_ok=True)

src.rename(dst)
```

In both examples, the file is moved from the source to the destination directory. `pathlib` is a bit more elegant and handles path manipulations in a straightforward way, which is why it's often the preferred approach in modern Python projects. Let me know if you want more examples or have specific use cases in mind!