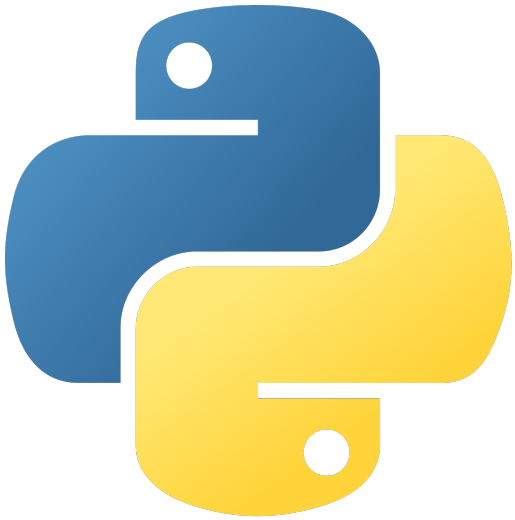
# Intermediate Python:
# Preparing for Data Structures and Algorithms

A document by
**Nathan Warner**



Computer Science
Northern Illinois University
United States
May 21, 2023

# Preface

Information for this document gathered from the elegant docementation page at:

https://book.pythontips.com/en/latest/

The purpose of this document is to delve into a wide range of intermediate Python concepts, aiming to offer a comprehensive understanding of the topic. It is assumed that you, as the reader, already possess a reasonable level of familiarity with Python at the beginner level, including a solid understanding of its syntax and fundamental programming concepts.

By building upon your existing knowledge, this document seeks to expand your proficiency in Python and equip you with the necessary skills to tackle more complex programming tasks. It covers various intermediate-level topics, including advanced data structures, object-oriented programming principles, file handling, error handling, and more.

Once an individual has grasped the concepts elucidated within this document, The subsequent logical progression would entail delving into the realm of data structures and algorithms.

## This Document Will NOT Include

- Variables and Simple Data Types
- Lists, Tuples, Dictionaries, Sets
- If Statements
- User Input and While Loops
- For Loops
- Basic Functions.

**Note:-**

It is advised that you are familliar with all Python built-in methods and functions.

- String: 47
- List: 11
- Tuple: 2
- Dict: 11
- Set: 17
- Functions: 68

# Contents

# 1 F-string formatting

**Formatting syntax:**

- :<char><<num> - Left-align the value within a field of width num.
- :<char>><num> - Right-align the value within a field of width num.
- :<char>^<num> - Center-align the value within a field of width num.
- :.<num> - Truncate strings to a maximum length of num characters.
- :.<num>f - Format floating-point numbers with num decimal places.
- :, - Add commas as thousands separator in integers and floating-point numbers.
- :_ - Add underscores as thousands separator in integers and floating-point numbers.

**Example:**

```
a = "nate"
print(f"{a:_^6}") # -> _nate_
```

## 2 Codecs Encoding

The codecs module in Python provides a way to encode and decode data, such as strings, using various character encodings. It allows you to convert text between different encodings, handle encoding and decoding errors, and perform transformations like base64 encoding or ROT13 encoding.

**Example**

```
#!/usr/bin/python
from codecs import encode
from codecs import decode

def main():
        a:str = "Nate"

        print(encode(a, "rot_13")) # -> Angr
        print(decode(a, "rot_13")) # -> Nate

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

# 3 Logging Ram Usage

To monitor memory usage in a standalone Python script, you can use other libraries or methods, such as psutil or memory_profiler. Here's an example using the psutil library:

Example Code:

```
#!/usr/bin/python3
from psutil import Process
from typing import List


def main() -> int:
    """ MAIN """
    process = Process()


    mylist: List[int] = list()
    for i in range(10000000):
        mylist.append(i)

    memory_usage = process.memory_info().rss / 1024 / 1024  # -> Memory usage: 396.5 MB
    print(f"Memory usage: {memory_usage} MB")



    return 0


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

# 4 Logging Execution Time

To time the execution of code in Python, you can use the time module.

Example Code:

```python
#!/usr/bin/python3

from time import time

def main() -> int:
    """ MAIN """

    start_time = time()

    mylist = []
    for i in range(100000000):
        mylist.append(i)


    end_time = time()

    execution_time = end_time - start_time

    print(f"Execution Time: {execution_time:.6f} Seconds")


    return 0


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

# 5 Unpacking Operator

In Python, the unpacking operator, represented by an asterisk (*), is used to unpack iterable objects such as lists, tuples, or dictionaries into individual elements. It allows you to extract the values from these objects and assign them to variables or use them as function arguments.

There are two main use cases for the unpacking operator:

1. Unpacking Iterables:

   Example:

   ```
   mylist: list[int] = [1,2,3]
   a, rest* = mylist
   print(a) # -> 1
   print(rest) # -> [2,3]
   ```

2. Unpacking Function Arguments: The unpacking operator can be used to pass multiple arguments to a function from an iterable object.

   Example:

   ```
   def my_function(a, b, c):
       print(a)
       print(b)
       print(c)

   my_list = [1, 2, 3]
   my_function(*my_list)
   # Prints:
   # 1
   # 2
   # 3
   ```

Additionally, the double asterisks (**) can be used to unpack a dictionary and pass its key-value pairs as keyword arguments to a function.

Example:

```
def my_function(name, age):
    print(f"Name: {name}")
    print(f"Age: {age}")

my_dict = {'name': 'John', 'age': 25}
my_function(**my_dict)
```

The unpacking operator provides a convenient way to work with iterable objects and simplifies the process of assigning values or passing arguments. It helps make your code more concise and readable.

# 6 *args and **kwargs

Now that we are familiar with the unpacking operator, we will now look at *args (variable arguments) and **kwargs (keyword arguments)

*args and **kwargs are special syntax used in function definitions to handle arbitrary numbers of arguments. They provide flexibility when the number of arguments passed to a function is unknown or can vary.

Combining *args and **kwargs allows you to define functions that can accept both positional and keyword arguments of varying lengths.

## 6.1 Usage of *args (sometimes called *argv)

*args is used to pass a variable number of non-keyword arguments to a function. The asterisk (*) before args allows you to pass multiple arguments as a tuple. Within the function, you can access these arguments using the args variable as if it were a tuple. This allows you to write functions that can accept any number of positional arguments.

Example:

```
def foo(*argv):
    for x,y in enumerate(argv):
        print(f"Argumunt {x}: {y}")

foo("nate", "warner")
```

Which will give the following output

```
Argument 0: nate
Argument 1: warner
```

## 6.2 Usage of **kwargs

**kwargs is used to pass a variable number of keyword arguments to a function. The double asterisks (**) before kwargs allow you to pass multiple keyword arguments as a dictionary. Within the function, you can access these arguments using the kwargs variable as if it were a dictionary. This allows you to write functions that can accept any number of keyword arguments.

Example:

```
def foo(**kwargs):
    for key, value in kwargs.items():
        print("{0} = {1}".format(key, value))

foo(name="nate", age="19")
```

**Note:-**

The names ”args” and ”kwargs” is just convention, they can be named anything. It’s the unpacking operator that is important.

# 7 Debugging (pdb) (Fuck Debugging)

Python provides several built-in debugging tools, such as the pdb module for interactive debugging, as well as integrated development environments (IDEs) with advanced debugging features. These tools make the debugging process more accessible and efficient, enabling you to identify and fix issues effectively, ultimately leading to better quality and more reliable code.

To be able to use *pdb* in your program.

```
from pdb import set_trace
```

From here we can introduce **set_trace()** anywhere in our code to set the starting point when commencing *pdb*, however, python 3.7 introduced a builtin function **breakpoint()**. This removes the need to import pdb and set a starting point with **set_trace()**. Now we can do the same with the **breakpoint()** function.

Once we have our **breakpoint()** or **set_trace()** function set in our code, we can run:

```
python3 -m pdb filename.py
```

To enter the debugger at our start point.

Once inside pdb, we have various commands at our disposal. If we run:

```
help
```

We can list all of the various commands. If we run:

```
list .
```

We will be able to see the lines of code **around** our breakpoint.

**Note:-**

The ' . ' after list is important, it will only show the lines near the breakpoint, whereas without the dot will keep going through your code.

Similarly,

```
ll
```

Will show a longer block of code.

To show where we are in execution, we can run:

```
where
```

To show variables, we can run:

```
locals()
globals()
p variable_name
pp name (pretty print)
```

Additionally, we can set breakpoints from within the debugger by running

```
break linenumber
```

Then if we run

```
continue (or just c)
```

It will jump to that breakpoint

# 8 Generators

An iterator is an object that enables a programmer to traverse a container, particularly lists. However, an iterator performs traversal and gives access to data elements in a container, but does not perform iteration. You might be confused so lets take it a bit slow. There are three parts namely:

- Iterable
- Iterator
- Iteration

## 8.1 Iterable

An iterable is any object in Python which has an ___iter___ or a ___getitem___ method defined which returns an iterator or can take indexes. In short an iterable is any object which can provide us with an iterator. So what is an iterator?

## 8.2 Iterator

An iterator is any object in Python which has a next (Python2) or ___next___ method defined. That's it. That's an iterator. Now let's understand iteration.

## 8.3 Iteration

In simple words it is the process of taking an item from something e.g a list. When we use a loop to loop over something it is called iteration. It is the name given to the process itself. Now as we have a basic understanding of these terms let's understand generators.

## 8.4 Generators

Generators are iterators, but you can only iterate over them once. It's because they do not store all the values in memory, they generate the values on the fly. You use them by iterating over them, either with a 'for' loop or by passing them to any function or construct that iterates. Most of the time generators are implemented as functions. However, they do not return a value, they yield it. Here is a simple example of a generator function:

```
def generator_function():
    for i in range(10):
        yield i

for item in generator_function():
    print(item)
```

Which will give the following output.

```
# Output: 0
# 1
# 2
# 3
# 4
# 5
# 6
# 7
# 8
# 9
```

The reason we use generators is because they require less resources.

Here is an example generator which calculates fibonacci numbers:

```
# generator version
def fibon(n):
    a = b = 1
    for i in range(n):
        yield a
        a, b = b, a + b

for x in fibon(1000000):
    print(x)
```

**Note:-**

The line a, b = b, a + b can be a bit confusing. Let me provide a clearer explanation:

Consider the scenario where a has a value of 5 and b has a value of 10. Intuitively, you might expect that b would be assigned the value of 20 since a is being updated to 10, leading to b becoming 10 + 10. However, this is not the case.

In reality, this line of code is executed in a single step, which means that the expression a + b does not take into account the new value of a. Instead, it uses the original value of 5.

Therefore, after executing this line, the values of a and b would be updated as follows:

- a receives the new value of 10.
- b is assigned the original value of a, which is 5.

So, despite the expectations, b does not receive the updated value of a. It's important to note that the assignments occur simultaneously, using the original values before any updates take place.

Rest assured, This behavior is consistent across programming languages because it adheres to a common principle of evaluating expressions and performing assignments in a well-defined order. It allows for predictable and consistent behavior in code execution.

Considering the fibonacci example, if we were to not use a generator, but instead house all numbers in a list, this would quite resource intensive, which is why using generators is a far better approach.

Furthermore, if we wanted to access the elements of a generator without using a loop, we can use Python's builtin function **next**

Example (Using fibonacci code):

```
x = fibon(10)
print(next(x)) -> 1
print(next(x)) -> 1
print(next(x)) -> 2
```

**Note:-**

Exhausting the generator with the next function will raise a **StopIteration** error. Using a for loop will not.

The last thing to mention is the process of turning an iterable into an iterator, to accomplish this we can you Python's builtin **iter** function.

# 9 Map, Filter and Reduce

These are three functions which facilitate a functional approach to programming. We will discuss them one by one and understand their use cases.

## 9.1 Map

The **map** function applies a function to all the items in an input list. Here is the blueprint:

Syntax:

```
map(function, iterable) -> rv: map object
```

So say we have a function:

```
def square(x):
    """Squares a number"""
    return x**2
```

Then, instead of calling this function with a single input $x$, we can use the **map** function to call this function on every element in an iterable.

Example:

```
mylist: List[int] = [2,4,6]
print(list(map(square, mylist))) -> [4,16,36]
```

**Note:-**

It's important we cast the map function to a list to view its contents, this is because the map function returns a **map object**.

Most of the times we use lambdas with map. Instead of a list of inputs we can even have a list of functions!

Example:

```
def multiply(x):
    return (x*x)
def add(x):
    return (x+x)

funcs = [multiply, add]
for i in range(5):
    value = list(map(lambda x: x(i), funcs))
    print(value)

# Output:
# [0, 0]
# [1, 2]
# [4, 4]
# [9, 6]
# [16, 8]
```

## 9.2 Filter

In Python, the **filter()** function is used to create a new list or iterator that contains elements from an iterable (like a list or tuple) for which a given function returns True.

Syntax:

```
filter(function, iterable) -> RV: Filter Object
```

Example Code:

```
def foo(x:int) -> bool:

    if x > 1:
        return True
    else:
        return False


mylist: list[int] = [0,1,2,3,4]

print(list(filter(foo, mylist)))
```

Which will ouput:

```
[2,3,4]
```

## 9.3 Reduce

In Python, the **reduce()** function is part of the **functools** module and is used to apply a specified function to a sequence of elements, progressively reducing it to a single value.

Syntax:

```
reduce(function, iterable)
```

Because this function is part of the **functools** modules, we must import it first.

```
from functools import reduce
```

Example Code:

```
from typing import List
from functools import reduce

def foo(x:int, y:int):

    return x+y

mylist: List[int] = [1,2,3,4,5]

print(reduce(foo, mylist)) # -> 15
```

# 10 Ternary Operators

In Python, the **ternary operator**, also known as the **conditional expression**, provides a concise way to write simple conditional statements in a single line. It allows you to evaluate an expression and return one of two values based on a given condition.

Syntax:

```
[value_if_true] if [condition] else [value_if_false]
```

Example Code:

```
a,b = 5,10
print("a" if a < b else "b") # -> a

a,b = 10,5
print("a" if a < b else "b") # -> b
```

Another more obscure and not widely used example involves tuples.

Syntax:

```
(if_test_false, if_test_true)[test]
```

Example:

```
dog = True
print(("no dog", "has dog")[dog]) # -> has dog
```

**Note:-**

The aforementioned illustration is not commonly employed and is generally disfavored among Python enthusiasts due to its lack of adherence to Pythonic principles. It also tends to cause confusion regarding the placement of the true and false values within the tuple. An additional rationale for steering clear of the use of a tupled ternary is that it necessitates the evaluation of both elements of the tuple, whereas the if-else ternary operator does not exhibit this behavior.

Example:

```
condition = True
print(2 if condition else 1/0)
#Output is 2

print((1/0, 2)[condition])
#ZeroDivisionError is raised
```

This happens because with the tupled ternary technique, the tuple is first built, then an index is found. For the if-else ternary operator, it follows the normal if-else logic tree. Thus, if one case could raise an exception based on the condition, or if either case is a computation-heavy method, using tuples is best avoided.

In python there is also the shorthand ternary tag which is a shorter version of the normal ternary operator you have seen above.

Example:

```
True or "Some" # -> True

False or "Some" # -> Some
```

This is helpful in case where you quickly want to check for the output of a function and give a useful message if the output is empty:

Or as a simple way to define function parameters with dynamic default values:

Example:

```
def foo(name:str, nickname=None) -> str:
    nickname = nickname or name
    return nickname


print(foo("nathan", "nate"))
print(foo("nathan"))
```

Another use case for the tenerary operator is handling empty strings

Example Code:

```
def foo(_str=""):

    return (_str  + "foo") if _str else "Null Input"


print(foo("nate")) # -> natefoo
print(foo()) # -> ""
```

# 11 Currying

Currying is a technique in functional programming where a function that takes multiple arguments is transformed into a sequence of functions, each taking a single argument. This allows you to partially apply the arguments to a function and create new specialized functions from the original one.

In Python, currying can be achieved using closures or higher-order functions. Here's a concise explanation of currying in Python:

**Example:**

```
def add(x):
    def inner(y):
        return x + y
    return inner

# Create a new function by partially applying add()
add2 = add(2)

# Call the new function with the remaining argument
result = add2(3)  # returns 5
```

**Example: Currying Double Function Call**

```
#!/usr/bin/python3

def foo(x):
    def bar(y):
        return x * y
    return bar


print(foo(2)(3)) # -> 6
```

**Note:-**

In the provided code, the foo() function returns the inner function bar(), which takes an argument y and returns the multiplication of x and y. The interesting part is that the returned bar() function is invoked immediately after calling foo(2) with (3) as its argument.

# 12 Decorators

Decorators play a crucial role in Python and can be defined as functions that alter the behavior of other functions. In essence, they contribute to making our code more concise and aligned with Pythonic practices. However, beginners often struggle to identify suitable scenarios for their usage. To address this, I will outline several areas where decorators can effectively streamline your code.

Undoubtedly, comprehending this concept can be challenging. Therefore, we will approach it incrementally to ensure a thorough understanding on your part.

## 12.1 Everything in Python is an object

First of all let's understand functions in Python:

In Python, functions are considered "first-class citizens," which means they are treated as any other object within the language. This concept allows functions to be assigned to variables, passed as arguments to other functions, returned as values from functions, and stored in data structures such as lists or dictionaries.

```
def foo(name="yasoob"):
    return "hello " + name

print(foo())
# output: 'hi nate'

# We can even assign a function to a variable like
greet = foo
# We are not using parentheses here because we are not calling the function foo
# instead we are just putting it into the greet variable. Let's try to run this

print(greet())
# output: 'hello nate'

# Let's see what happens if we delete the old hi function!
del foo
print(foo())
#outputs: NameError

print(greet())
#outputs: 'hi nate'
```

## 12.2 Defining functions within functions

In Python we can define functions inside other functions:

## 12.3 Returning functions from within functions

Now we need to learn one more thing, that functions can return functions too.

It is not necessary to execute a function within another function, we can return it as an output as well:

Example:

```
def foo(name="nate"):
    def bar():
        return "now you are in the greet() function"
```

```
    def foobar():
        return "now you are in the welcome() function"

    if name == "nate":
        return bar
    else:
        return foobar

a = foo()
print(a)
#outputs: <function bar at 0x7f2143c01500>

#This clearly shows that `a` now points to the bar() function in foo()
#Now try this

print(a())
#outputs: now you are in the bar() function
```

## 12.4 Giving a function as an argument to another function

Example:

```
def hi():
    return "hi nate!"

def doSomethingBeforeHi(func):
    print("I am doing some boring work before executing hi()")
    print(func())

doSomethingBeforeHi(hi)
#outputs:I am doing some boring work before executing hi()
#           hi nate!
```

## 12.5 Writing your first decorator

Now we can use all the information learned previously and construct a decorator

Example Code:

```
#!/usr/bin/python3

def a_new_decorator(a_func):

    def wrapTheFunction():
        print("I am doing some boring work before executing a_func()")

        a_func()

        print("I am doing some boring work after executing a_func()")

    return wrapTheFunction

def a_function_requiring_decoration():
    print("I am the function which needs some decoration to remove my foul smell")

foo = a_new_decorator(a_function_requiring_decoration)
```

```
# now a_function_requiring_decoration is wrapped by wrapTheFunction()


foo()
#outputs:I am doing some boring work before executing a_func()
#        I am the function which needs some decoration to remove my foul smell
#        I am doing some boring work after executing a_func()
```

Now you might be wondering why we did not use the @ anywhere in our code? That is just a short way of making up a decorated function. Here is how we could have run the previous code sample using @.

```
def a_new_decorator(a_func):

    def wrapTheFunction():
        print("I am doing some boring work before executing a_func()")

        a_func()

        print("I am doing some boring work after executing a_func()")

    return wrapTheFunction

@a_new_decorator
def a_function_requiring_decoration():
    print("I am the function which needs some decoration to remove my foul smell")


a_function_requiring_decoration()
#outputs:I am doing some boring work before executing a_func()
#        I am the function which needs some decoration to remove my foul smell
#        I am doing some boring work after executing a_func()
```

**Note:-**

the @a_new_decorator is just a short way of saying:
foo = a_new_decorator(a_function_requiring_decoration)

However, there is a problem with our code, if we call:

```
print(a_function_requiring_decoration.__name__)
```

We will see that we get an output of wrapTheFunction. To fix this, we can use **functools functools.wraps**

```
from functools import wraps

def a_new_decorator(a_func):
    @wraps(a_func)
    def wrapTheFunction():
        print("I am doing some boring work before executing a_func()")
        a_func()
        print("I am doing some boring work after executing a_func()")
    return wrapTheFunction

@a_new_decorator
def a_function_requiring_decoration():
    """Hey yo! Decorate me!"""
    print("I am the function which needs some decoration to "
          "remove my foul smell")

print(a_function_requiring_decoration.__name__)
# Output: a_function_requiring_decoration
```

**Note:-**

Note: @wraps takes a function to be decorated and adds the functionality of copying over the function name, docstring, arguments list, etc. This allows us to access the pre-decorated function's properties in the decorator.

## 12.6 Decorators with Arguments

Additionally, we can have decorators that require arguments, this requires a another nested function.

Example Code:

```
#!/usr/bin/python3
from functools import wraps

def function(x=5):
    def foo(f):
        @wraps(f)
        def bar(a,y):
            return(f"x = {x} a={a}, y={y}, a+y = {f(a,y)}")

        return bar
    return foo



@function(12)
def foobar(a,y):
    return a+y

print(foobar(1,2)) # -> x = 12, a = 1, y = 2, a+y = 3
```

## 12.7 Decorator Classes

A decorator class is defined by creating a class that implements the ___call___ method. This method is invoked when an instance of the class is called as a function or used to decorate another function or class. The ___call___ method receives the original function or class as its first argument, and you can modify or wrap it as needed.

Example Code:

```
#!/usr/bin/python3

class foo(object):

    x = 12

    def __init__(self,func):
        self.func=func

    def __call__(self, a,b):
        print(self.x)

        return self.func(1,2)

@foo
def foobar(a,b):
    return a+b

print(foobar(1,2)) # -> 12 \n 3
```

# 13 Global & Return

Now we will examine the **global** keyword in python. Global variable means that we can access that variable outside the scope of the function as well. Let me demonstrate it with an example:

```
# first without the global variable
def add(value1, value2):
    result = value1 + value2

add(2, 4)
print(result)

# Oh crap, we encountered an exception. Why is it so?
# the python interpreter is telling us that we do not
# have any variable with the name of result. It is so
# because the result variable is only accessible inside
# the function in which it is created if it is not global.
Traceback (most recent call last):
  File "", line 1, in
    result
NameError: name 'result' is not defined

# Now lets run the same code but after making the result
# variable global
def add(value1, value2):
    global result
    result = value1 + value2

add(2, 4)
result
6
```

**Note:-**

In practical programming you should try to stay away from global keyword as it only makes life difficult by introducing unwanted variables to the global scope.

## 13.1 Multiple return values

So what if you want to return two variables from a function instead of one? There are a couple of approaches which new programmers take. The most famous approach is to use global keyword. Let's take a look at a useless example:

Example Code:

```
def profile():
    global name
    global age
    name = "Danny"
    age = 30

profile()
print(name) # -> Danny

print(age) # -> 30
```

Some try to solve this problem by returning a tuple, list or dict with the required values after the function terminates. It is one way to do it and works like a charm:

Example Code:

```
def profile():
    name = "Danny"
    age = 30
    return (name, age)

profile_data = profile()
print(profile_data[0])
# Output: Danny

print(profile_data[1])
# Output: 30
```

Or by more common convention:

```
def profile():
    name = "Danny"
    age = 30
    return name, age

profile_name, profile_age = profile()
print(profile_name)
# Output: Danny
print(profile_age)
# Output: 30
```

Keep in mind that even in the above example we are returning a tuple (despite the lack of paranthesis) and not separate multiple values. If you want to take it one step further, you can also make use of namedtuple. Here is an example:

```
from collections import namedtuple
def profile():
    Person = namedtuple('Person', 'name age')
    return Person(name="Danny", age=31)

# Use as namedtuple
p = profile()
print(p, type(p))
# Person(name='Danny', age=31) <class '__main__.Person'>
print(p.name)
# Danny
print(p.age)
#31

# Use as plain tuple
p = profile()
print(p[0])
# Danny
print(p[1])
#31

# Unpack it immediatly
name, age = profile()
print(name)
# Danny
print(age)
#31
```

**Note:-**

This is a better way to do it along with returning lists and dicts. Don't use global keyword unless you know what you are doing. global might be a better option in a few cases but is not in most of them.

## 14 Nonlocal

The nonlocal keyword is used to work with variables inside nested functions, where the variable should not belong to the inner function.

Use the keyword *nonlocal* to declare that the variable is not local.

```
def outer():
    x = "Nate"
    def inner():                          """
        nonlocal x                        Outputs
        x = "Hello"                       Hello
    inner()                               """
    return x
print(outer())
```

# 15 Mutation

Table 1: Class Descriptions and Immutability

| Class | Description | Immutable? |
|---|---|---|
| bool | Boolean value | Yes |
| int | Integer (arbitrary magnitude) | Yes |
| float | Floating-point number | Yes |
| list | Mutable sequence of objects | No |
| tuple | Immutable sequence of objects | Yes |
| str | Character string | Yes |
| set | Unordered set of distinct objects | No |
| frozenset | Immutable form of set class | Yes |
| dict | Associative mapping (aka dictionary) | No |

The mutable and immutable datatypes in Python cause a lot of headache for new programmers. In simple words, mutable means 'able to be changed' and immutable means 'constant'. Want your head to spin? Consider this example:

```
foo = ['hi']
print(foo)
# Output: ['hi']

bar = foo
bar += ['bye']
print(foo)
# Output: ['hi', 'bye']
```

It's not a bug. It's mutability in action. Whenever you assign a variable to another variable of mutable datatype, any changes to the data are reflected by both variables. The new variable is just an alias for the old variable. This is only true for mutable datatypes.

**Note:-**

Deleting bar using the **del** statement will not delete foo even though they refer to the same object in memory. The deletion of bar will solely **remove the reference to foo**, but the actual object foo will **remain unaffected**.

Here is a gotcha involving functions and mutable data types:

```
def add_to(num, target=[]):
    target.append(num)
    return target

add_to(1)
# Output: [1]
```

```
add_to(2)
# Output: [1, 2]

add_to(3)
# Output: [1, 2, 3]
```

You might have expected it to behave differently. You might be expecting that a fresh list would be created when you call add_to like this:

```
def add_to(num, target=[]):
    target.append(num)
    return target

add_to(1)
# Output: [1]

add_to(2)
# Output: [2]

add_to(3)
# Output: [3]
```

Well again it is the mutability of lists which causes this pain. In Python the default arguments are evaluated once when the function is defined, not each time the function is called. You should never define default arguments of mutable type unless you know what you are doing. You should do something like this:

```
def add(x):
    _list: List[int] = list()
    _list.append(x)
    return _list

print(add(12)) # -> [12]
print(add(13)) # -> [13]
print(add(14)) # -> [14]
```

Now whenever you call the function without the target argument, a new list is created.

# 16 ___slots___ Magic

In Python every class can have instance attributes. By default Python uses a dict to store an object's instance attributes. This is really helpful as it allows setting arbitrary new attributes at runtime.

However, for small classes with known attributes it might be a bottleneck. The dict wastes a lot of RAM. Python can't just allocate a static amount of memory at object creation to store all the attributes. Therefore it sucks a lot of RAM if you create a lot of objects (I am talking in thousands and millions). Still there is a way to circumvent this issue. It involves the usage of ___slots___ to tell Python not to use a dict, and only allocate space for a fixed set of attributes. Here is an example with and without ___slots___:

Class without slots:

```
class MyClass(object):
    def __init__(self, name, identifier):
        self.name = name
        self.identifier = identifier
```

With slots:

```
class MyClass(object):
    __slots__ = ['name', 'identifier']
    def __init__(self, name, identifer):

        self.name = name
        self.identifer = identifer
```

# 17 Virtual Environment

**Virtualenv** is a tool which allows us to make isolated python environments. Imagine you have an application that needs version 2 of a library, but another application requires version 3. How can you use and develop both these applications?

If you install everything into /usr/lib/python2.7/site-packages (or whatever your platform's standard location is), it's easy to end up in a situation where you unintentionally upgrade a package.

In another case, imagine that you have an application which is fully developed and you do not want to make any change to the libraries it is using but at the same time you start developing another application which requires the updated versions of those libraries.

What will you do? Use virtualenv! It creates isolated environments for your python application and allows you to install Python libraries in that isolated environment instead of installing them globally.

To install virtualenv:

```
pip install virtualenv
```

To start a virtual environment:

```
virtualenv name
source name/bin/activate
```

Or for us fish users:

```
virtualenv name
source name/bin/activate.fish
```

While creating the virtualenv you have to make a decision. Do you want this virtualenv to use packages from your system site-packages or install them in the virtualenv's site-packages? By default, virtualenv will not give access to the global site-packages.

If you want your virtualenv to have access to your systems site-packages, use the `--system-site-packages` switch when creating your virtualenv like this:

```
virtualenv --system-site-packages mycoolproject
```

We can **Deactivate** the virtualenv with:

```
deactivate
```

Some other python virtual environments worth mentioning:

- **venv:** Venv is a built-in module in Python 3 and serves as the recommended way to create virtual environments. It provides similar functionality to virtualenv but is included in the Python standard library.
- **pipenv:** Pipenv is a higher-level tool that combines package management with virtual environment creation. It simplifies managing dependencies and environments by automatically creating and managing both the virtual environment and the associated packages.
- **conda:** Conda is a popular cross-platform package and environment management system. It allows you to create virtual environments, manage package installations, and handle dependencies for not only Python but also other programming languages.
- **pyenv:** Pyenv is a tool for managing multiple Python versions on your system. While it's primarily focused on managing different Python installations, it also allows you to create isolated virtual environments for each Python version.

# 18 Collections

Python ships with a module that contains a number of container data types called Collections. We will talk about a few of them and discuss their usefulness.

The ones which we will talk about are:

- **defaultdict**
- **OrderedDict**
- **Counter**
- **deque**
- **namedtuple**
- **enum.Enum** (outside of the module; Python 3.4+)

## 18.1 defaultdict

The Python defaultdict type behaves almost exactly like a regular Python dictionary, but if you try to access or modify a missing key, then defaultdict will automatically create the key and generate a default value for it. This makes defaultdict a valuable option for handling missing keys in dictionaries.

Here are the differences between pythons dict and collections default dict:

- **Default Values:** The main distinction is how they handle missing keys. In a regular Python dict, if you try to access a key that doesn't exist, it raises a KeyError. On the other hand, a defaultdict allows you to define a default value that will be returned when accessing a non-existent key. This default value is specified when creating the defaultdict.
- **Initialization:** When creating a defaultdict, you need to provide a callable object (e.g., a function or lambda expression) that will be used to generate the default values. This callable will be invoked without arguments and should return the default value type you desire. In contrast, a regular dict is initialized without any default value specification.
- **Automatic Key Insertion:** When accessing a non-existent key in a defaultdict, it will automatically insert the key into the dictionary along with the default value generated by the callable. This behavior is different from a regular dict, where accessing a missing key raises a KeyError without modifying the dictionary.

*Syntax:*

```
from collections import defauldict
defauldict(default_factory)
```

With traditional dictionaries, if we did something like:

```
a = (
    ('Yasoob', 'Yellow'),
    ('Ali', 'Blue'),
    ('Arham', 'Green'),
    ('Ali', 'Black'),
    ('Yasoob', 'Red'),
    ('Ahmed', 'Silver'),
)
# a = (
#         ("nate", "warner"),
#         ("foo", "bar"),
```

```
    #         ("foofoo", "barbar")
    # )

    mydict = dict()
    for x,y in a:
        mydict[x] =y
    print(mydict) # -> {'Yasoob': 'Red', 'Ali': 'Black', 'Arham': 'Green', 'Ahmed': 'Silver'}
```

We will see that the color "Yellow" has been overwritten. This is because "Yellow" and "Red" have the same key. To account for this, we would have to do something like:

To Account for this, we would have to make our dict values as lists and do something like:

```
        mydict = dict()
    for x,y in a:
        if x not in mydict:
            mydict[x] =y
    print(mydict) -> {'Yasoob': 'Yellow', 'Ali': 'Blue', 'Arham': 'Green', 'Ahmed': 'Silver'}


    mydict = dict()
    for x,y in a:
        if x in mydict.keys():
            mydict[x].append(y)
        else:
            mydict[x] = [y]
    print(mydict) # ->
{'Yasoob': ['Yellow', 'Red'], 'Ali': ['Blue', 'Black'], 'Arham': ['Green'], 'A
hmed': ['Silver']}
```

with defaultdict you do not need to check whether a key is present or not. So we can do:

```
from collections import defaultdict

colours = (
    ('Yasoob', 'Yellow'),
    ('Ali', 'Blue'),
    ('Arham', 'Green'),
    ('Ali', 'Black'),
    ('Yasoob', 'Red'),
    ('Ahmed', 'Silver'),
)

favourite_colours = defaultdict(list)

for name, colour in colours:
    favourite_colours[name].append(colour)

print(favourite_colours)

# output
# defaultdict(<type 'list'>,
#    {'Arham': ['Green'],
#     'Yasoob': ['Yellow', 'Red'],
#     'Ahmed': ['Silver'],
#     'Ali': ['Blue', 'Black']
# })
```

## 18.2 OrderedDict

OrderedDict is a dictionary subclass in Python that remembers the order in which items were added. In a regular Python dictionary, the order of the items is not guaranteed, and it may change between different runs of the program or different versions of Python. However, an OrderedDict preserves the order of the items as they were added, even if new items are later added or existing items are changed.

OrderedDict is part of the collections module in Python. It provides all the methods and functionality of a regular dictionary, as well as some additional methods that take advantage of the ordering of the items. Here are some examples of using OrderedDict in Python:

```
from collections import OrderedDict
a = OrderedDict([('a',1), ('b',2), ('c',3)])
a['d'] = 4

print(a)
for x,y in a.items():
    print(x,y)
```

## 18.3 Counter

A Counter is a dict subclass for counting hashable objects. It is a collection where elements are stored as dictionary keys and their counts are stored as dictionary values. Counts are allowed to be any integer value including zero or negative counts. The Counter class is similar to bags or multisets in other languages.

Elements are counted from an iterable or initialized from another mapping (or counter):

```
c = Counter()
c = Counter('gallahad')
c = Counter({'red': 4, 'blue': 2})
c = Counter(cats=4, dogs=8)
```

```
c = Counter(['eggs', 'ham'])
c['bacon'] # -> 0
```

Counter objects have a dictionary interface except that they return a zero count for missing items instead of raising a KeyError:

Setting a count to zero does not remove an element from a counter. Use del to remove it entirely:

```
c['sausage'] = 0
del c['sausage']
```

**Counter methods (Counter objects inherit dictionary methods, those listed are additional):**

- `elements()`: Return an iterator over elements repeating each as many times as its count
- `most_common([n])`: Return a list of the n most common elements and their counts from the most common to the least. If n is omitted or None, most_common() returns all elements in the counter. Elements with equal counts are ordered in the order first encountered
- `subtract([iterable-or-mapping])`: Elements are subtracted from an iterable or from another mapping (or counter)
- `total()`: Compute the sum
- `update([iterable-or-mapping])`:
- `+` (addition)
- `-` (subtraction)
- `&` (intersection)
- `|` (union)

## 18.4 namedtuple

You might already be acquainted with tuples. A tuple is basically a immutable list which allows you to store a sequence of values separated by commas. They are just like lists but have a few key differences. The major one is that unlike lists, you can not reassign an item in a tuple. In order to access the value in a tuple you use integer indexes like:

```
mytuple: Tuple[str,int] = ("foo", 12)
print(mytuple[0]) # -> foo
```

Well, so now what are namedtuples? They turn tuples into convenient containers for simple tasks. With namedtuples you don't have to use integer indexes for accessing members of a tuple. You can think of namedtuples like dictionaries but unlike dictionaries they are immutable.

```
#!/usr/bin/python3
from collections import namedtuple

animal = namedtuple("Animal", ["name", "age", "type"])
dog = animal(name="charles", age="2", type="brown")
print(dog) # Output -> Animal(name='charles', age='2', type='brown')
print(dog.name) # Output -> charles
```

# 19 Enumerate

enumerate is a built-in function in Python that adds a counter to an iterable object, such as a list or a string. It returns an iterator that yields pairs containing the index and the corresponding item from the iterable.

Example Code:

```
#!/usr/bin/python3
from typing import List

def main() -> int:
    """ MAIN """

    a: List[int] = [8,98,987,897,976]

    for x in enumerate(a):
        print(x,end=", ")


    return 0

if __name__ == '__main__':
    main()
# Output ->
```

And there is more! enumerate also accepts an optional argument that allows us to specify the starting index of the counter.

```
#!/usr/bin/python3
from typing import List

def main() -> int:

    mylist: List[int] = [5,82,28,2763,23782]

    for x in enumerate(mylist, 1):
        print(x, end=", ")

    return 0

if __name__ == '__main__':
    main()

# Output -> (1, 5), (2, 82), (3, 28), (4, 2763), (5, 23782)
```

## 20 Zip and unzip

Zip is a useful function that allows you to combine two lists easily.

After calling zip, an iterator is returned. In order to see the content wrapped inside, we need to first convert it to a list.

```
first_name = ['Joe','Earnst','Thomas','Martin','Charles']

last_name = ['Schmoe','Ehlmann','Fischer','Walter','Rogan','Green']

age = [23, 65, 11, 36, 83]

print(list(zip(first_name,last_name, age)))

# Output
#
# [('Joe', 'Schmoe', 23), ('Earnst', 'Ehlmann', 65), ('Thomas', 'Fischer', 11),
('Martin', 'Walter', 36), ('Charles', 'Rogan', 83)]
```

We can also use the **zip** function to **unzip** a list aswell This time, we need an input of a list with an asterisk before it.

```
full_name_list = [('Joe', 'Schmoe', 23),
                  ('Earnst', 'Ehlmann', 65),
                  ('Thomas', 'Fischer', 11),
                  ('Martin', 'Walter', 36),
                  ('Charles', 'Rogan', 83)]

first_name, last_name, age = list(zip(*full_name_list))
print(f"first name: {first_name}\nlast name: {last_name} \nage: {age}")

# Output

# first name: ('Joe', 'Earnst', 'Thomas', 'Martin', 'Charles')
# last name: ('Schmoe', 'Ehlmann', 'Fischer', 'Walter', 'Rogan')
# age: (23, 65, 11, 36, 83)
```

# 21 Object introspection

In computer programming, introspection is the ability to determine the type of an object at runtime. It is one of Python's strengths. Everything in Python is an object and we can examine those objects. Python ships with a few built-in functions and modules to help us.

## 21.1 dir

In this section we will learn about dir and how it facilitates us in introspection.

It is one of the most important functions for introspection. It returns a list of attributes and methods belonging to an object. Here is an example:

```
my_list = [1, 2, 3]
dir(my_list)
# Output: ['__add__', '__class__', '__contains__', '__delattr__', '__delitem__',
# '__delslice__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
# '__getitem__', '__getslice__', '__gt__', '__hash__', '__iadd__', '__imul__',
# '__init__', '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__',
# '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__',
# '__setattr__', '__setitem__', '__setslice__', '__sizeof__', '__str__',
# '__subclasshook__', 'append', 'count', 'extend', 'index', 'insert', 'pop',
# 'remove', 'reverse', 'sort']
```

Our introspection gave us the names of all the methods of a list. This can be handy when you are not able to recall a method name. If we run dir() without any argument then it returns all names in the current scope.

## 21.2 type and id

The type function returns the type of an object. For example:

```
print(type(''))
# Output: <type 'str'>

print(type([]))
# Output: <type 'list'>

print(type({}))
# Output: <type 'dict'>

print(type(dict))
# Output: <type 'type'>

print(type(3))
# Output: <type 'int'>
```

id returns the unique ids of various objects. For instance:

```
name = "Yasoob"
print(id(name))
# Output: 139972439030304
```

## 21.3 inspect module

The inspect module also provides several useful functions to get information about live objects. For example you can check the members of an object by running:

```
import inspect
print(inspect.getmembers(str))
# Output: [('__add__', <slot wrapper '__add__' of ... ...
```

# 22 Comprehensions

Comprehensions are constructs that allow sequences to be built from other sequences. Several types of comprehensions are supported in both Python 2 and Python 3:

## 22.1 list comprehensions

List comprehensions provide a short and concise way to create lists. It consists of square brackets containing an expression followed by a for clause, then zero or more for or if clauses. The expressions can be anything, meaning you can put in all kinds of objects in lists. The result would be a new list made after the evaluation of the expression in context of the if and for clauses.

Syntax for list comprehension without else clause

```
variable = [expression for condition if condition]
```

Syntax For list comprehension with else clause:

```
variable = [expression if condition else condition for condition]
```

Example:

```
multiples = [i for i in range(30) if i % 3 == 0]
print(multiples)
# Output: [0, 3, 6, 9, 12, 15, 18, 21, 24, 27]
```

**Note:-**

Notice that the **for clause** goes at the end iff you need an **else clause**, otherwise **if** condition goes at the end.

List comprehension with double for loop.

**Example:**

```
return sorted(set([word for word in a1 for word_2 in a2 if word in word_2]))
```

Is the same as:

```
myset = set()
for word in a1:
    for word_2 in a2:
        if word in word_2:
            myset.add(word)
return sorted(myset)
```

## 22.2 dict comprehensions

Similar to list comprehensions, we can prefrom **dict comprehensions** with curly braces instead of square brackets

Example:

```
mcase = {'a': 10, 'b': 34, 'A': 7, 'Z': 3}

mcase_frequency = {
    k.lower(): mcase.get(k.lower(), 0) + mcase.get(k.upper(), 0)
    for k in mcase.keys()
}

# mcase_frequency == {'a': 17, 'z': 3, 'b': 34}
```

You can also quickly switch keys and values of a dictionary:

```
{v:k for k,v in some_dict.items()}
```

Example:

```
 #!/usr/bin/python3
from typing import Dict

def count(s:str) -> Dict[str,int]:
    hashmap: Dict[str,int] = dict()
    hashmap =  {char:s.count(char) for char in s} if s else {}

    return hashmap

def main() -> int:
    """ MAIN """

    print(count("aba"))
    return 0


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

## 22.3 set comprehensions

Set comprehensions are the same as list comprehensions, except they use curly braces.

Example:

```
myset = {x**2 for x in [1,1,2]}
Output -> {1,4}
```

## 22.4 generator comprehensions

Generator comprehensions are also similar to list comprehensions. The only difference is that they don't allocate memory for the whole list but generate one item at a time, thus more memory efficient.

Example:

```
mygen = [x for x in range(30) if x %2 == 0 and x !=0]
for i in mygen:
    print i
Output ->
2
4
6
...
```

# 23 Exceptions

Exception handling is an art which once you master grants you immense powers. I am going to show you some of the ways in which we can handle exceptions.

In basic terminology we are aware of the try/except structure. The code that can cause an exception to occur is put in the try block and the handling of the exception is implemented in the except block. The code in the except block will only execute if the try block runs into an exception. Here is a simple example:

Example:

```
mydict = {1:2, 2:4}
try:
    print(mydict[3])
except KeyError:
    print("Not a key in dict")
```

## 23.1 Handling multiple exceptions

However it is possible to handle **multiple exceptions.**

Example:

```
try:
    file = open('test.txt', 'rb')
except (IOError, EOFError) as e:
    print("An error occurred. {}".format(e.args[-1]))
```

Another method is to handle individual exceptions in separate except blocks. We can have as many except blocks as we want. Here is an example:

Example:

```
try:
    file = open('test.txt', 'rb')
except EOFError as e:
    print("An EOF error occurred.")
    raise e
except IOError as e:
    print("An error occurred.")
    raise e
```

This way if the exception is not handled by the first except block then it may be handled by a following block, or none at all. Now the last method involves trapping ALL exceptions:

Example:

```
try:
    file = open('test.txt', 'rb')
except Exception as e:
    # Some logging if you want
    raise e
```

**Note:-**

catching all exceptions may have unintended consequences because catching all exceptions may also catch the ones you want to occur; for example, in many command-line based programs, pressing control+c will terminate the program, but if you catch all excepts, the KeyboardInterrupt will be caught as an exception, so pressing control+c will NOT terminate the program.

## 23.2 Finally Clause

We wrap our main code in the try clause. After that we wrap some code in an except clause which gets executed if an exception occurs in the code wrapped in the try clause. In this example we will use a third clause as well which is the finally clause. The code which is wrapped in the finally clause will run whether or not an exception occurred. It might be used to perform clean-up after a script. Here is a simple example:

```
try:
    file = open('test.txt', 'rb')
except IOError as e:
    print('An IOError occurred. {}'.format(e.args[-1]))
finally:
    print("This would be printed whether or not an exception occurred!")

# Output: An IOError occurred. No such file or directory
# This would be printed whether or not an exception occurred!
```

## 23.3 Try/Else Clause

Often times we might want some code to run if no exception occurs. This can easily be achieved by using an else clause. One might ask: why, if you only want some code to run if no exception occurs, wouldn't you simply put that code inside the try? The answer is that then any exceptions in that code will be caught by the try, and you might not want that. Most people don't use it and honestly I have myself not used it widely. Here is an example:

```
try:
    print('I am sure no exception is going to occur!')
except Exception:
    print('exception')
else:
    # any code that should only run if no exception occurs in the try,
    # but for which exceptions should NOT be caught
    print('This would only run if no exception occurs. And an error here '
          'would NOT be caught.')
finally:
    print('This would be printed in every case.')

# Output: I am sure no exception is going to occur!
# This would only run if no exception occurs. And an error here would NOT be caught
# This would be printed in every case.
```

**Note:-**

The else clause would only run if no exception occurs and it would run before the finally clause.

# 24 Classes

Classes are the core of Python. They give us a lot of power but it is really easy to misuse this power. In this section I will share some obscure tricks and caveats related to classes in Python. Let's get going!

## 24.1 New style classes

New style classes were introduced in Python 2.1 but a lot of people do not know about them even now! It is so because Python also supports old style classes just to maintain backward compatibility. I have said a lot about new and old but I have not told you about the difference. Well the major difference is that:

- Old base classes do not inherit from anything
- New style base classes inherit from object

A very basic example is:

```
class OldClass():
    def __init__(self):
        print('I am an old class')

class NewClass(object):
    def __init__(self):
        print('I am a jazzy new class')

old = OldClass()
# Output: I am an old class

new = NewClass()
# Output: I am a jazzy new class
```

This inheritance from object allows new style classes to utilize some magic. A major advantage is that you can employ some useful optimizations like ___slots___. You can use super() and descriptors and the likes. Bottom line? Always try to use new-style classes.

**Note:-**

Python 3 only has new-style classes. It does not matter whether you subclass from object or not. However it is recommended that you still subclass from object.

## 24.2 Members

In Python classes, members refer to the attributes and methods defined within the class. Here's a concise explanation of different types of class members:

- **Instance variables:** These are variables that are specific to each instance of a class. They hold data that is unique to each object and can be accessed and modified using the instance of the class. Instance variables are typically defined within the class's methods using the self parameter.
- **Class variables:** Class variables are shared among all instances of a class. They are defined within the class but outside any methods. Class variables hold data that is common to all objects of the class. These variables are accessed using the class name itself.
- **Methods:** Methods are functions defined within a class that can perform specific actions or operations on the class's data. They are associated with the class and can access both instance variables and class variables. Methods are typically defined with the def keyword and the first parameter is conventionally named self to refer to the instance of the class.
- **Class methods:** Class methods are methods that operate on the class itself rather than instances of the class. They are defined using the @classmethod decorator and have access to the class and its class variables.
- **Static methods:** Static methods are methods that do not have access to the instance or class variables. They are defined using the @staticmethod decorator and can be called on the class or an instance. Static methods are generally used for utility functions or operations that don't require access to the state of the class.

These different types of class members provide flexibility and encapsulation within a class, allowing you to define and manipulate data and behaviors according to the requirements of your program.

## 24.3 Instance & Class variables

Most beginners and even some advanced Python programmers do not understand the distinction between instance and class variables. Their lack of understanding forces them to use these different types of variables incorrectly. Let's understand them.

The basic difference is:

- Instance variables are for data which is unique to every object
- Class variables are for data shared between different instances of a class

There are not many issues while using immutable class variables. This is the major reason due to which beginners do not try to learn more about this subject because everything works! If you also believe that instance and class variables can not cause any problem if used incorrectly then check the next example.

```
class SuperClass(object):
    superpowers = []

    def __init__(self, name):
        self.name = name

    def add_superpower(self, power):
        self.superpowers.append(power)

foo = SuperClass('foo')
bar = SuperClass('bar')
foo.name
```

```
# Output: 'foo'

bar.name
# Output: 'bar'

foo.add_superpower('fly')
bar.superpowers
# Output: ['fly']

foo.superpowers
# Output: ['fly']
```

That is the beauty of the wrong usage of mutable class variables. To make your code safe against this kind of surprise attacks then make sure that you do not use mutable class variables. You may use them only if you know what you are doing.

## 24.4 Private

In Python, private class members are typically denoted by using a naming convention that starts with a double underscore (___). This convention indicates that the variable or method is intended for internal use within the class and should not be accessed directly from outside the class. Here's a concise explanation of private class members:

- **Naming convention:** Private class members are named with a double underscore prefix, such as ___variable or ___method(). This convention is known as name mangling and indicates that the member is intended to be private.
- **Encapsulation:** Private class members are used to encapsulate data and behavior within the class, preventing direct access and modification from outside the class. They help enforce encapsulation and protect the internal implementation of the class.
- **Limited accessibility:** Private members are not accessible outside the class by their original names. Python automatically modifies the name of private variables by adding a prefix _ClassName to avoid naming conflicts. For example, ___variable becomes _ClassName___variable.

**Note:-**

It's important to note that the double underscore naming convention is a convention rather than a strict enforcement by the language. It serves as a signal to other developers that the member should be treated as private, but it is still technically possible to access private members using the modified name.

Here's a simple example to illustrate private class members:

```
class MyClass:
    def __init__(self):
        self.__private_var = 42

    def __private_method(self):
        return "Private method"

obj = MyClass()
print(obj._MyClass__private_var)  # Output: 42
print(obj._MyClass__private_method())  # Output: Private method
```

## 24.5 Protected

In Python, a protected member is a class member (variable or method) that is intended to be accessible within the class itself and its subclasses. It is denoted by using a single underscore (_) as a prefix before the member name.

The use of a single underscore (_) as a convention indicates that the member should be treated as "protected" and should not be accessed directly from outside the class or its subclasses. However, unlike some other programming languages, Python does not enforce strict access control, and it is still possible to access and modify protected members from outside the class.

## 24.6 Private vs Protected

Private Members:

- Private members are denoted by a double underscore (___) as a prefix before the member name.
- Private members can only be accessed within the class itself, and they are not directly accessible from its subclasses or from outside the class.
- Python performs name mangling to make private members harder to access, by adding a class-specific prefix to the member name. However, this is more of a name-mangling convention than strict access control.
- The use of private members provides a higher level of encapsulation and emphasizes that the member is intended to be internal and not accessed or modified from outside the class.

Protected Members:

- Protected members are denoted by a single underscore (_) as a prefix before the member name.
- Protected members can be accessed within the class itself and its subclasses.
- However, they can still be accessed and modified from outside the class, although it is considered a convention to treat them as "protected" and not access them directly.
- The use of protected members allows for a limited level of encapsulation and provides a way to communicate that a member should be treated as internal or semi-private.

## 24.7 Public

In Python, public class members are variables and methods that are intended to be accessible and usable from outside the class. They are not designated with any special naming conventions and can be accessed directly. Here's a concise explanation of public class members:

## 24.8 ___Init___ Function (Constructor)

The ___init___ function in Python is a special method that is automatically called when an object is created from a class. It is commonly known as the constructor method. Here's a concise explanation of the ___init___ function:

The ___init___ function allows you to initialize the attributes or variables of an object when it is created. It defines the initial state of the object by assigning values to its instance variables. It is called with the self parameter (which refers to the instance being created) along with any additional parameters you want to pass when creating the object.

Example Code:

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

person1 = Person("John", 25)
print(person1.name)  # Output: John
print(person1.age)   # Output: 25
```

## 24.9 Self Parameter

The self parameter is a reference to the instance of the class on which the method is called. It allows the method to access and manipulate the instance variables and other methods of the class.

## 24.10 Destructer

Similar to the constructor, we can build a **Destructer**

**Example:**

```
#!/usr/bin/python3
class foo(object):
    def __init__(self, name):
      self.name = name

    def __del__(self):
        print("Deleted Object")

a = foo()
del a # -> deleted object
```

**Note:-**

We will also see "Deleted Object" even if we dont call **del** on the object, because all the objects are deleted at the end of execution.

## 24.11 ___Str___ / ___repr___

The ___str___ function in Python is a special method that provides a string representation of an object. It allows you to define how an object should be represented as a string when using the built-in str() or print() functions.

By implementing the ___str___ method in a class, you can customize the textual representation of objects to provide meaningful information about their state or characteristics. This method should return a string that represents the object in a human-readable format.

Example Code:

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

    def __str__(self):
        return f"Person: {self.name}, Age: {self.age}"

person = Person("John", 25)
print(person)  # Output: Person: John, Age: 25
```

In the example above, the ___str___ method is defined in the Person class. It returns a formatted string that represents the object's name and age attributes. When the print() function is called on the person object, it automatically invokes the ___str___ method to obtain the string representation and displays it.

The ___repr___() method returns a more information-rich, or official, string representation of an objec

## 24.12 Instance Method

Class instance methods in Python are functions defined within a class that operate on instances of the class. They have access to the instance-specific data and can manipulate it. Here's a concise explanation of class instance methods:

- **Definition:** Class instance methods are defined using the **def** keyword within a class without any special decorators. They are automatically passed the instance as the first parameter, conventionally named **self**, allowing them to access and manipulate instance variables.
- **Access to instance variables:** Class instance methods have access to the instance-specific data, including instance variables. They can read and modify these variables using the self parameter.
- **Behavior specific to instances:** Class instance methods define behaviors or actions that are specific to instances of the class. They can perform operations based on the state or attributes of the instance they are called upon.
- **Usage:** Class instance methods are typically called on instances of the class. They can access and modify instance-specific data, invoke other instance methods, or perform actions relevant to the specific instance.
- **Common scenarios:** Class instance methods are commonly used to encapsulate behaviors that involve the state of individual instances. They can perform operations on instance variables, calculate derived attributes, update the state of the instance, or interact with other objects.

Example Code:

```
class MyClass:
    def __init__(self, value):
        self.value = value

    def increment(self):
        self.value += 1

    def get_value(self):
        return self.value

obj = MyClass(10)
obj.increment()
print(obj.get_value())  # Output: 11
```

## 24.13 Class Methods

Python class methods are functions defined within a class that operate on the class itself or its instances. They encapsulate behaviors and actions that objects of the class can perform. Here's a concise explanation of class methods:

- **Definition:** Class methods are defined using the def keyword within a class and are typically decorated with @classmethod to indicate their special nature.
- **Access to class and class variables:** Class methods have access to the class itself via the cls parameter, allowing them to manipulate class-level variables and perform actions that affect the class as a whole.
- **Alternative constructors:** Class methods are commonly used as alternative constructors. By defining a class method that creates and returns instances of the class with specific configurations, you can provide multiple ways to create objects.
- **No direct access to instance variables:** Class methods don't have access to the instance-specific data (instance variables) unless explicitly passed as arguments.
- **Usage:** Class methods can be called either on the class itself or on instances of the class. When called on the class, they affect the class-level state, while when called on instances, they can interact with both class-level and instance-specific data.
- **Common scenarios:** Class methods are often used for utility functions that don't require instance-specific data, operations that involve class-level variables, or operations that need to create instances with specific configurations.

Example Code:

```
class MyClass:
    class_variable = 0

    def __init__(self, value):
        self.instance_variable = value

    @classmethod
    def increment_class_variable(cls):
        cls.class_variable += 1

    def get_instance_variable(self):
        return self.instance_variable
```

## 24.14 Static Method

Python class static methods are methods defined within a class that do not have access to the class or its instances. They are self-contained and operate independently of the class's state. Here's a concise explanation of class static methods:

- **Definition:** Class static methods are defined using the @staticmethod decorator before the method definition. They do not take any special parameters like the class or instance, meaning they cannot access or modify class or instance variables.
- **Independence from class and instances:** Static methods are self-contained and operate independently of the class or its instances. They do not require access to instance-specific or class-level data.
- **Utility functions:** Static methods are commonly used for utility functions that are related to the class but do not require access to instance or class-specific data. They can perform calculations, conversions, or other operations that are relevant to the class but do not involve its state.
- **No access to self or cls:** Static methods cannot access instance-specific data or class-level data through the self or cls parameters. They can only work with the parameters passed explicitly to them.
- **Usage:** Static methods can be called on the class itself or on instances of the class. They are often used when a particular functionality is closely associated with the class but does not require access to its state.

Example:

```
class MathUtils:
    @staticmethod
    def add_numbers(a, b):
        return a + b

result = MathUtils.add_numbers(3, 5)
print(result)  # Output: 8
```

## 24.15 Inheritance

Inheritance is a fundamental concept in object-oriented programming (OOP) that allows one class to inherit the properties and methods of another class. In Python, classes can be defined in such a way that they inherit characteristics from other classes, referred to as the parent or base class.

The class that inherits from another class is called the child or derived class. By inheriting from a parent class, the child class automatically gains access to all the attributes (variables) and methods defined in the parent class. This enables code reuse and promotes the creation of a hierarchical structure for organizing related classes.

To establish inheritance in Python, you define the parent class and then create the child class with a declaration that includes the parent class name in parentheses. This is done by specifying the parent class as an argument to the child class definition.

Example:

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

    def speak(self):
        print("An animal speaks.")

class Dog(Animal):
    def speak(self):
        print("The dog barks.")

dog = Dog("Buddy")
print(dog.name)  # Output: Buddy
dog.speak()      # Output: The dog barks.
```

In this example, we have a parent class called Animal that defines an ___init___ method and a speak method. The child class Dog inherits from Animal by including it in parentheses during its class definition.

The child class Dog has its own speak method, which overrides the speak method inherited from Animal. When we create an instance of Dog and call the speak method, it executes the overridden method from the child class, printing "The dog barks."

Furthermore, the child class Dog also inherits the ___init___ method from Animal, allowing us to initialize the name attribute when creating a Dog instance.

If we wanted to build an ___init___ method for the **Dog** class, we can call upon the **super** method.

Example:

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

    def speak(self):
        print("An animal speaks.")

class Dog(Animal):

    def __init__(self,name,_type):
        super().__init__(name)
        self._type = _type
```

```
    def speak(self):
        print("The dog barks.")

dog = Dog("Buddy", "Lab")
print(dog.name)  # Output: Buddy
dog.speak()      # Output: The dog barks.
print(dog._type)
```

As you can see, in the Dog class, we defined an ___init___ method that has all the parameters of the parent ___init___ method, and adds the parameters that are specific to the Dog class at the end. Then, in the constructor, we call **super.___init___()** with the parameter that is in the Animal constructor. Notice we dont need self here.

## 24.16 Encapsulation

Python encapsulation is a principle of object-oriented programming (OOP) that allows data and methods to be bundled together within a class, and restricts access to them from outside the class. It helps in achieving data abstraction and data hiding, providing a way to protect the internal state of an object from direct modification or access.

Encapsulation in Python is implemented by using access modifiers: public, protected, and private. Public members (variables and methods) are accessible from anywhere within the program. Protected members can be accessed within the class and its subclasses. Private members, on the other hand, are only accessible within the class itself.

By using access modifiers, encapsulation provides a mechanism to control the visibility and accessibility of data and methods, preventing unintended modifications or misuse. This promotes better code organization, improves code maintainability, and reduces the likelihood of errors by enforcing data integrity and encapsulated behavior.

## 24.17 Polymorphism

Polymorphism in Python is the ability of an object to take on multiple forms or have multiple behaviors. It allows different objects to be treated as if they were of the same type, providing flexibility and reusability in code.

In Python, polymorphism is achieved through method overriding and method overloading. Method overriding occurs when a subclass provides a different implementation of a method that is already defined in its superclass. This allows objects of different classes to have the same method name but behave differently.

Method overloading, on the other hand, is the ability to define multiple methods with the same name but different parameters. Python does not natively support method overloading based on the number or type of parameters, but it can be achieved using default parameter values or using libraries like functools or multipledispatch.

Polymorphism helps simplify code by allowing objects to be treated uniformly, regardless of their specific types. It promotes code reuse, modularity, and flexibility in object-oriented programming.

**Polymorphism Example: (Method Overriding)**

```
class Shape:
    def area(self):
        pass

class Rectangle(Shape):
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def area(self):
        return self.length * self.width

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

    def area(self):
        return 3.14 * self.radius**2

# Create objects of different shapes
rectangle = Rectangle(5, 3)
circle = Circle(4)

# Call the area method on different objects
print(rectangle.area())  # Output: 15
print(circle.area())     # Output: 50.24
```

## 24.18 Method Overloading

In Python, method overloading is not directly supported as it is in some other programming languages. However, we can achieve similar behavior by using default parameter values or by using libraries like **functools** or **multipledispatch**. Here's an example using default parameter values:

**Example:**

```
class MathOperations:
    def add(self, a, b, c=None):
        if c is not None:
            return a + b + c
        else:
            return a + b
# Create an instance of MathOperations
math = MathOperations()

# Call the add method with different numbers of arguments
print(math.add(2, 3))       # Output: 5
print(math.add(2, 3, 4))    # Output: 9
```

We can also achieve **overloading** with functools **singledispatch** decorator

**Example:**

```python
#!/usr/bin/python3
from functools import singledispatch, singledispatchmethod

class MathOperations:
    @singledispatchmethod
    def add(self, *args):
        raise NotImplementedError("Unsupported argument types")

    @add.register(int)
    def add_int(self,*args):
        result = 0
        for item in args:
            result+=item
        return result

    @add.register(str)
    def add_str(self,*args):
        result = ""
        for item in args:
            result+=item + " "
        return result

    @add.register(list)
    def add_list(self,*args):
        result = []
        for item in args:
            result += item
        return result



math = MathOperations()
try:
    print(math.add(2, 3))                #5
    print(math.add("Hello", "World"))    #Hello World
    print(math.add([1, 2], [3, 4]))      #[1, 2, 3, 4]
    print(math.add(2.5, 3.7))            #Not implemented, Unsupported argument types
except NotImplementedError as E:
    print(f"Not implemented, {E}")
```

# 25 Magic Methods

Python magic methods, also known as special methods or dunder methods (short for "double underscore" methods), are predefined methods that provide functionality to classes in Python. These methods are invoked implicitly in response to specific events or operations, such as object creation, attribute access, arithmetic operations, and more.

Magic methods are denoted by double underscores (___) at the beginning and end of their names. For example, ___init___, ___str___, and ___add___ are some commonly used magic methods.

**Example for ___getitem___ method**

```
class GetTest(object):
    def __init__(self):
        self.info = {
            'name':'Yasoob',
            'country':'Pakistan',
            'number':12345812
        }

    def __getitem__(self,i):
        return self.info[i]

foo = GetTest()

foo['name']
# Output: 'Yasoob'

foo['number']
# Output: 12345812
```

Without the ___getitem___ method we would have got this error:

```
>>> foo['name']

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'GetTest' object has no attribute '__getitem__'
```

Which means traditionally we would have to do something like

```
foo.info["name"]
```

## 25.1 Operator Overloading (For Classes)

Now that we know what magic methods are, lets see how we can use them for **operator overloading.**

**Example:**

```
class Vector(object):
    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(10,20)
v2 = Vector(50,60)
v3 = v1+v2
print(v3.x, v3.y) #-> 60, 80
```

In addition to ___add___, there are several other dunder methods available for Python operators.

## 25.2 ___repr___ and ___str___

With the dunder method **repr**, we change change the behavior of a print call with an object. Let's consider the vector class from operator overloading

**Example: Repr**

Let's consider what would happen if we attempt to print the object *v3*

```
print(v3) # -> <__main__.Vector object at 0x7f341108be10>
```

However, if we define the ___repr___ method, we can change this behavior.

```
#!/usr/bin/python3
class Vector(object):
    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 __repr__(self):
        return f"x: {self.x}, y: {self.y}"

v1 = Vector(10,20)
v2 = Vector(50,60)
v3 = v1+v2
print(v3) # -> x: 60, y: 80
```

**Note:-**

We can get the same result if instead of ___repr___ we used ___str___

# 26 Asynchronous Programming in Python

**Asynchronous programming** in Python allows you to write concurrent and non-blocking code, making it possible to perform multiple tasks concurrently without waiting for each task to complete before moving on to the next one. It is particularly useful when dealing with I/O-bound operations, such as network requests or file operations, where there is a significant amount of waiting involved.

Python provides an asynchronous programming framework called **asyncio**, which is built on top of coroutines, event loops, and futures. Here's a basic overview of the main concepts:

- **Coroutines:** Coroutines are special functions that can be paused and resumed. They are defined using the **async** keyword before the function definition. Coroutines allow you to write asynchronous code in a sequential manner.
- **async and await keywords:** The async keyword is used to define a coroutine, and the await keyword is used to pause the execution of a coroutine until a certain asynchronous operation completes. The await keyword is used with functions or methods that return awaitable objects.
- **Event Loop:** An event loop is an essential component of an asynchronous program. It is responsible for scheduling and executing coroutines. The event loop manages multiple coroutines by running them concurrently and switching between them when they are blocked.
- **Futures:** Futures represent the results of asynchronous operations. They are placeholders for the eventual results and can be used to check if an operation has completed or to retrieve its result.

**Example:**

```
#!/usr/bin/python3
import asyncio

async def greet():
    print("Hello")
    await asyncio.sleep(1)  # Simulate an I/O-bound operation
    print("World")

async def main():
    await asyncio.gather(greet(), greet(), greet())  # Execute coroutines concurrently

asyncio.run(main())
```

**Note:-**

This program executes **all** three **greet** function calls at the same time.

## 26.1 Coroutines

An asyncio coroutine is a special function that is designed to perform asynchronous operations without blocking the event loop

Coroutines declared with the async/await syntax is the preferred way of writing asyncio applications. For example, the following snippet of code prints “hello”, waits 1 second, and then prints “world”:

**Example:**

```
import asyncio
import time

async def say_after(delay, what):
    await asyncio.sleep(delay)
    print(what)

async def main():
    print(f"started at {time.strftime('%X')}")

    await say_after(1, 'hello')
    await say_after(2, 'world')

    print(f"finished at {time.strftime('%X')}")

asyncio.run(main())
```

**Note:-**

Note that simply calling a coroutine will not schedule it to be executed:

## 26.2 Event Loop

The asyncio **event loop** is a core component of the asyncio module in Python. It provides an **execution environment** for asyncio coroutines and tasks, allowing them to be scheduled and executed **asynchronously**.

The **event loop** manages the execution of **coroutines, callbacks, and I/O operations** in an event-driven manner. It keeps track of pending tasks and determines which tasks are ready to be executed based on their state.

**Example:**

```
import asyncio

async def my_coroutine():
    print("Coroutine is starting")
    await asyncio.sleep(3)
    print("Coroutine has finished")

loop = asyncio.get_event_loop()
loop.run_until_complete(my_coroutine())
```

**Note:-**

In many cases, using **asyncio.gather()** or **asyncio.create_task()** is a more convenient and recommended way to work with coroutines and tasks in asyncio instead of directly interacting with the **event loop**.

## 26.3 Gather

You may notice that in the above code snippet the coroutines are executed sequentially, not concurrently.

Here, the program waits for 1 second to print ”hello”, then waits an additional 2 seconds to print ”world”. The total execution time will be around 3 seconds.

If you want the coroutines to run concurrently, you can use **asyncio.gather()** to await them together. Here's an example:

**Example:**

```
import asyncio
import time

async def say_after(delay, what):
    await asyncio.sleep(delay)
    print(what)

async def main():
    print(f"started at {time.strftime('%X')}")

    await asyncio.gather(
        say_after(1, "Hello"),
        say_after(2, "World")
    )

    print(f"finished at {time.strftime('%X')}")

asyncio.run(main())
```

## 26.4 Tasks, asyncio.create_task(*coro, name=None, context=None*)

Alternatively, rather than using **asyncio.gather()**, The asyncio.create_task() function to run coroutines concurrently as asyncio Tasks.

**Example:**

```
import asyncio

async def say_after(delay, what)
      await asyncio.sleep(delay)
      print(what)

async def main():
    task1 = asyncio.create_task(
        say_after(1, 'hello'))

    task2 = asyncio.create_task(
        say_after(2, 'world'))

    print(f"started at {time.strftime('%X')}")

    # Wait until both tasks are completed (should take
    # around 2 seconds.)
    await task1
    await task2

    print(f"finished at {time.strftime('%X')}")
```

## 26.5 Task Groups

The **asyncio.TaskGroup** class provides a more modern alternative to **create_task()**

**Example:**

```
#!/usr/bin/python3
import asyncio

async def do_one():
    print("Calculating Return Value")
    await asyncio.sleep(1)
    print(10)

async def do_other():
    print("Calculating Return Value")
    await asyncio.sleep(1)
    print(5)


async def main():

    async with asyncio.TaskGroup() as tg:
        task1 = tg.create_task(do_one()) # -> outputs 10
        task2 = tg.create_task(do_other()) # -> outputs 5


asyncio.run(main())
```

## 26.6 Task Cancellation

**Example:**

```
#!/usr/bin/python3
import asyncio

async def do_stuff():
    print("Doing Stuff")
    await asyncio.sleep(3)
    print("Did stuff")

async def main():
    task1 = asyncio.create_task(do_stuff())

    await asyncio.sleep(2)
    task1.cancel()

    try:
        await task1
    except asyncio.CancelledError:
        print("Canceled Task")

asyncio.run(main())
```

## 26.7 Return Values

**With Gather Example:**

```
#!/usr/bin/python3
import asyncio
import time

async def do_one():
    print("Calculating Return Value")
    await asyncio.sleep(1)
    return 10

async def do_other():
    print("Calculating Return Value")
    await asyncio.sleep(1)
    return 5


async def main():

    a = await asyncio.gather(do_one(), do_other()) # -> [10,5]

    for item in a:
        print(item)

asyncio.run(main())
```

**With Tasks Example:**

```
#!/usr/bin/python3
import asyncio

async def do_one():
    print("Calculating Return Value")
    await asyncio.sleep(1)
    return 10

async def do_other():
    print("Calculating Return Value")
    await asyncio.sleep(1)
    return 5

async def main():
    task1 = asyncio.create_task(do_one())
    task2 = asyncio.create_task(do_other())

    a = await task1, await task2

    for item in a:
        print(item, end=" ")

asyncio.run(main())
```

## 26.8 Gather VS Tasks

**Functionality:**

- **asyncio.gather(*awaitable1, awaitable2*)**: It is a utility function that takes multiple awaitable objects (coroutines, tasks, or futures) and schedules them to run concurrently. It returns a single awaitable object that represents the collection of results from all the tasks.
- **asyncio.create_task(*coroutine*)** It is a class that represents an individual coroutine wrapped as a task. It can be scheduled to run concurrently and provides more control over the execution, including cancellation and exception handling.

**Return Values:**

- **asyncio.gather(*awaitable1, awaitable2*):** It returns a single awaitable object that represents the collection of results from all the tasks. You can await this object to retrieve the results when all the tasks are completed.
- **asyncio.create_task(*coroutine*):** When a task is created, it returns the task object immediately. You can use this object to monitor the status of the task and retrieve its result when it's completed.

**Error Handling:**

- **asyncio.gather(*awaitable1, awaitable2*):** By default, if any of the tasks passed to asyncio.gather() raises an exception, the other tasks will continue to run. The raised exception will be propagated and can be retrieved when awaiting the asyncio.gather() result.
- **asyncio.create_task(*coroutine*):** Each task can have its own exception handling mechanism. You can use try/except blocks within the coroutine or use the add_done_callback() method of the task to handle exceptions raised during execution.

Overall, asyncio.gather() is more suitable when you want to schedule multiple tasks concurrently and retrieve their results together. On the other hand, asyncio.Task provides more fine-grained control over individual tasks, allowing you to monitor and handle exceptions specific to each task.

## 26.9 Futures

A **Future** is a special low-level **awaitable object** that represents an eventual result of an **asynchronous operation**.

When a **Future object** is awaited it means that the **coroutine** will wait until the Future is resolved in some other place.

**Future objects** in asyncio are needed to allow callback-based code to be used with async/await.

Normally there is no need to create **Future objects** at the application level code.

**Example:**

```
#!/usr/bin/python3
import asyncio

async def do_stuff():
    print("Doing STuff")
    await asyncio.sleep(2)
    print("Did stuff")
    return 10

loop = asyncio.get_event_loop()
future = asyncio.ensure_future(do_stuff())

def callback(future):
    print(f"Return value: {future.result()}")

future.add_done_callback(callback)
loop.run_until_complete(future)
```

## 26.10 Timeouts

An asynchronous context manager that can be used to limit the amount of time spent waiting on something.

delay can either be None, or a float/int number of seconds to wait. If delay is None, no time limit will be applied; this can be useful if the delay is unknown when the context manager is created.

In either case, the context manager can be rescheduled after creation using Timeout.reschedule().

**Example:**

```
#!/usr/bin/python3
import asyncio

async def do_stuff():
    print("Doing Stuff")
    await asyncio.sleep(3)
    print("Did stuff")

async def main():
    try:
        async with asyncio.timeout(1):
            await do_stuff()

    except asyncio.TimeoutError:
        print("Timeout")

asyncio.run(main())
```

## 26.11 Asynchronous Iterators

First let's acknowledge the difference between an asynchronous iterator and a normal iterator.

The key difference between an async iterator and a normal iterator is how they handle iteration and the nature of their ___iter___ and ___next___ methods.

The main difference can be summarized as follows:

**Normal Iterator:**

- Synchronous iteration.
- Implements ___iter___ and ___next___ methods.
- The ___next___ method returns the next item synchronously.
- Raises StopIteration to indicate the end of iteration.

**Async Iterator:**

- Asynchronous iteration.
- Implements ___aiter___ and ___anext___ methods.
- The ___anext___ method returns the next item asynchronously.
- Can pause the execution of the coroutine using await until the next item is available.
- Raises StopAsyncIteration to indicate the end of iteration.

When we say that an async iterator is used to iterate over a collection asynchronously, it means that the iteration process can occur concurrently and does not block the execution of other tasks.

In summary, iterating over a collection asynchronously means that the iteration process is non-blocking and allows for concurrent execution of other tasks, resulting in improved efficiency, responsiveness, and better utilization of system resources, especially in scenarios involving I/O-bound operations.

**Example:**

```
#!/usr/bin/python3
import asyncio

class foo(object):
    def __init__(self,start,stop):
        self.start= start
        self.stop = stop
        self.current = self.start

    async def __aenter__(self):
        print(f"Entering Iterator at Start Value {self.start}...")
        return self

    async def __aexit__(self,exc_type,exc_val,exc_tb):
        print(f"Exiting Iterator at Exit Value {self.stop} (not inclusive)...")
        if exc_type is not None:
            print(f"{exc_type}, {exc_val}")

    def __aiter__(self):
        return self

    async def __anext__(self):
        if self.current < self.stop:
            value = self.current
            self.current += 1
            return value
        else:
            raise StopAsyncIteration

async def do_stuff():
    print("Doing stuff")
    await asyncio.sleep(2)
    print("did stuff")

async def use_iterator():
    async with foo(0,5) as thing:
        async for item in thing:
            print(item)

async def main():
    await asyncio.gather(do_stuff(), use_iterator())

asyncio.run(main())
"""
Outputs:
Doing stuff
Entering Iterator at Start Value 0...
0
1
2
3
4
```

```
Exiting Iterator at Exit Value 5 (not inclusive)...
did stuff
"""
```

**Note:-**

So you can notice that we run our iterator and our do_stuff function concurrently

## 26.12 Asynchronous Looping

The async loop allows you to perform continuous tasks asynchronously while concurrently executing other tasks, enabling efficient utilization of resources and responsiveness in asynchronous programming.

**Example:**

```python
#!/usr/bin/python3
import asyncio

async def my_coroutine():
    while True:
        print("Async loop")
        await asyncio.sleep(1)

async def main():
    # Start the async loop
    task = asyncio.create_task(my_coroutine())

    # Perform other tasks concurrently
    for i in range(5):
        print(f"Concurrent task {i}")
        await asyncio.sleep(0.5)

    # Cancel the async loop
    task.cancel()

asyncio.run(main())
```

# 27 Threading

## 27.1 What is threading?

A thread is a separate execution path, allowing your program to perform multiple tasks seemingly simultaneously. However, in most Python 3 implementations, threads don't truly run concurrently. While it might seem like different processors are working independently, they're actually taking turns.

Although threading might resemble having multiple processors executing tasks at the same time, it's not the case due to Python's Global Interpreter Lock (GIL). Only one Python thread can run at a time, which can limit performance gains. Threading is suitable for tasks involving waiting for external events but might not accelerate CPU-intensive tasks significantly.

Python code running on the standard CPython implementation adheres to these limitations. Threads written in C can bypass the GIL and run concurrently. For CPU-bound tasks using standard Python, consider exploring the multiprocessing module.

Adopting threading can enhance program design even if it doesn't always boost performance. Many examples in this tutorial leverage threads not primarily for speed but for improved clarity and comprehension in program structure.

## 27.2 Starting a Thread:

The Python standard library provides threading, which contains most of the primitives you'll see in this article. Thread, in this module, nicely encapsulates threads, providing a clean interface to work with them.

```
#!/usr/bin/python3
import threading
from time import sleep

def func1(name):
    print("do stuff from function %s" % (name))
    sleep(1)
    print("Did stuff from function %s" % (name))
def func2(name):
    print("doing other stuff from function %s" % (name))
    sleep(1)
    print("Did other stuff from function %s" % (name))

thread1 = threading.Thread(target=func1, args=(1,))
thread2 = threading.Thread(target=func2, args=(2,))
thread1.start()
thread2.start()
"""
Output:
do stuff from function 1
doing other stuff from function 2
Did stuff from function 1
Did other stuff from function 2
"""
```

## 27.3 Daemon Threads

In computer science, a daemon is a process that runs in the background.

Python threading has a more specific meaning for daemon. A daemon thread will shut down immediately when the program exits. One way to think about these definitions is to consider the daemon thread a thread that runs in the background without worrying about shutting it down.

If a program is running Threads that are not daemons, then the program will wait for those threads to complete before it terminates. Threads that are daemons, however, are just killed wherever they are when the program is exiting.

```
#!/usr/bin/python3
import threading
from time import sleep

def func1(name):
    print("do stuff from function %s" % (name))
    sleep(1)
    print("Did stuff from function %s" % (name))

def func2(name):
    print("doing other stuff from function %s" % (name))
    sleep(1)
    print("Did other stuff from function %s" % (name))


thread1 = threading.Thread(target=func1, args=(1,), daemon=True)
thread2 = threading.Thread(target=func2, args=(2,),daemon=True)

thread1.start()
thread2.start()

"""
Output:
do stuff from function 1
doing other stuff from function 2
"""
```

## 27.4 .join() a Thread

To tell one thread to wait for another thread to finish, you call .join().

```
 #!/usr/bin/python3
import threading
from time import sleep

def func1(name):
    print("do stuff from function %s" % (name))
    sleep(1)
    print("Did stuff from function %s" % (name))

def func2(name):
    print("doing other stuff from function %s" % (name))
    sleep(1)
    print("Did other stuff from function %s" % (name))


thread1 = threading.Thread(target=func1, args=(1,), daemon=True)
thread2 = threading.Thread(target=func2, args=(2,),daemon=True)

thread1.start()
thread1.join()
thread2.start()

"""
Output:
do stuff from function 1
Did stuff from function 1
doing other stuff from function 2
"""
```

## 27.5 Working With Many Threads

Frequently, you'll want to start a number of threads and have them do interesting work. Let's start by looking at the harder way of doing that, and then you'll move on to an easier method.

The harder way of starting multiple threads is the one you already know:

```
#!/usr/bin/python3
import threading
from time import sleep

def func1(name):
    print("do stuff from function %s" % (name))
    sleep(1)
    print("Did stuff from function %s" % (name))

threads = []
for i in range(1,4):
    x = threading.Thread(target=func1, args=(i,))
    threads.append(x)
    x.start()

"""
Output:
do stuff from function 1
do stuff from function 2
do stuff from function 3
Did stuff from function 1
Did stuff from function 2
Did stuff from function 3
"""
```

And we can use the join function instead if we want to wait on the functions:

```
#!/usr/bin/python3
import threading
from time import sleep

def func1(name):
    print("do stuff from function %s" % (name))
    sleep(1)
    print("Did stuff from function %s" % (name))

threads = []
for i in range(1,4):
    x = threading.Thread(target=func1, args=(i,))
    threads.append(x)
    # x.start()

for i,j in enumerate(threads):
    j.start()
    j.join()
```

## 27.6 Using a ThreadPoolExecutor

There's an easier way to start up a group of threads than the one you saw above. It's called a ThreadPoolExecutor, and it's part of the standard library in concurrent.futures (as of Python 3.2).

The easiest way to create it is as a context manager, using the with statement to manage the creation and destruction of the pool.

```
#!/usr/bin/python3
import threading
import concurrent.futures
from time import sleep

def func1(name):
    print("do stuff from function %s" % (name))
    sleep(1)
    print("Did stuff from function %s" % (name))

threads = []

with concurrent.futures.ThreadPoolExecutor(max_workers=3) as executer:
    executer.map(func1,range(1,4))

"""
Output:
do stuff from function 1
do stuff from function 2
do stuff from function 3
Did stuff from function 1
Did stuff from function 2
Did stuff from function 3
"""
```

The code creates a ThreadPoolExecutor as a context manager, telling it how many worker threads it wants in the pool. It then uses .map() to step through an iterable of things, in your case range(3), passing each one to a thread in the pool.

The end of the with block causes the ThreadPoolExecutor to do a .join() on each of the threads in the pool. It is strongly recommended that you use ThreadPoolExecutor as a context manager when you can so that you never forget to .join() the threads.

## 27.7 Race Conditions

Race conditions can occur when two or more threads access a shared piece of data or resource.

## 27.8 Basic Synchronization Using Lock

There are a number of ways to avoid or solve race conditions. You won't look at all of them here, but there are a couple that are used frequently. Let's start with Lock.

To solve your race condition above, you need to find a way to allow only one thread at a time into the read-modify-write section of your code. The most common way to do this is called Lock in Python. In some other languages this same idea is called a mutex. Mutex comes from MUTual EXclusion, which is exactly what a Lock does.

```
#!/usr/bin/python3
import threading
import concurrent.futures
from time import sleep

class foobar(object):
    def __init__(self):
        self.x = 0
        self._lock = threading.Lock()
    def do_stuff(self, name):
        print(f"{name} about to lock")
        with self._lock:
            print(f"{name} has lock")
            sleep(1)
            self.x+=1
            print(f"Updated Value, {name} releasing lock")
        print(f"Relased lock")

f1 = foobar()
with concurrent.futures.ThreadPoolExecutor(max_workers=3) as executer:
    executer.map(f1.do_stuff, range(1,4))
print(f"Ending value: {f1.x}")

"""
Output:
1 about to lock
1 has lock
2 about to lock
3 about to lock
Updated Value, 1 releasing lock
Relased lock
2 has lock
Updated Value, 2 releasing lock
Relased lock
3 has lock
Updated Value, 3 releasing lock
Relased lock
Ending value: 3
"""
```

## 27.9 Threading Objects

There are a few more primitives offered by the Python threading module. While you didn't need these for the examples above, they can come in handy in different use cases, so it's good to be familiar with them.

### Semphore

The first Python threading object to look at is threading.Semaphore. A Semaphore is a counter with a few special properties. The first one is that the counting is atomic. This means that there is a guarantee that the operating system will not swap out the thread in the middle of incrementing or decrementing the counter.

The internal counter is incremented when you call .release() and decremented when you call .acquire().

The next special property is that if a thread calls .acquire() when the counter is zero, that thread will block until a different thread calls .release() and increments the counter to one.

Semaphores are frequently used to protect a resource that has a limited capacity. An example would be if you have a pool of connections and want to limit the size of that pool to a specific number.

### Timer

A threading.Timer is a way to schedule a function to be called after a certain amount of time has passed. You create a Timer by passing in a number of seconds to wait and a function to call:

```
import threading
def do():
    print(f"doing stuff")

l = threading.Timer(5, do)
l.start()
```

So this program will wait for 5 seconds before calling the function

If you want to stop a Timer that you've already started, you can cancel it by calling .cancel(). Calling .cancel() after the Timer has triggered does nothing and does not produce an exception.

### Barrier

A threading.Barrier can be used to keep a fixed number of threads in sync. When creating a Barrier, the caller must specify how many threads will be synchronizing on it. Each thread calls .wait() on the Barrier. They all will remain blocked until the specified number of threads are waiting, and then the are all released at the same time.

Remember that threads are scheduled by the operating system so, even though all of the threads are released simultaneously, they will be scheduled to run one at a time.

One use for a Barrier is to allow a pool of threads to initialize themselves. Having the threads wait on a Barrier after they are initialized will ensure that none of the threads start running before all of the threads are finished with their initialization.

# 28 Lambdas

Lambdas are one line functions. They are also known as anonymous functions in some other languages. You might want to use lambdas when you don't want to use a function twice in a program. They are just like normal functions and even behave like them.

*Syntax:*

```
identifier = lambda parameters: expression
```

*Example:*

```
add5 = lambda x: x+5
print(add5(10)) # -> 15
```

*Example: List sorting*

```
a = [(1, 2), (4, 1), (9, 10), (13, -3)]
a.sort(key=lambda x: x[1])

print(a)
# Output: [(13, -3), (4, 1), (1, 2), (9, 10)]
```

# 29 for/else

For loops have an else clause which most of us are unfamiliar with. The else clause executes after the loop completes normally. This means that the loop did not encounter a break statement. They are really useful once you understand where to use them. I, myself, came to know about them a lot later.

*Example:*

```
        for item in container:
    if search_something(item):
        # Found it!
        process(item)
        break
else:
    # Didn't find anything..
    not_found_in_container()
```

# 30 Coroutines

Coroutines are similar to generators with a few differences. The main differences are:

- generators are data producers
- coroutines are data consumers

First of all let's review the generator creation process. We can make generators like this:

```
def fib():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a+b
for item in fib():
    print(item)
```

It is fast and does not put a lot of pressure on memory because it generates the values on the fly rather than storing them in a list. Now, if we use yield in the above example, more generally, we get a coroutine. Coroutines consume values which are sent to it. A very basic example would be a grep alternative in Python:

```
def grep(pattern):
    print("Searching for", pattern)
    while True:
        line = (yield)
        if pattern in line:
            print(line)
```

Wait! What does yield return? Well we have turned it into a coroutine. It does not contain any value initially, instead we supply it values externally. We supply values by using the .send() method. Here is an example:

```
search = grep('coroutine')
next(search)
# Output: Searching for coroutine
search.send("I love you")
search.send("Don't you love me?")
search.send("I love coroutines instead!")
# Output: I love coroutines instead!
search = grep('coroutine')
# ...
search.close()
```

# 31 Function caching

Function caching allows us to cache the return values of a function depending on the arguments. It can save time when an I/O bound function is periodically called with the same arguments. Before Python 3.2 we had to write a custom implementation. In Python 3.2+ there is an lru_cache decorator which allows us to quickly cache and uncache the return values of a function.

## 31.1 Python 3.2+

```
from functools import lru_cache

@lru_cache(maxsize=32)
def foo(n):
    match n:
        case 0:
            return 0
        case 1 | 2:
            return 1
        case other:
            return foo(n-1) + foo(n-2)

print(foo(100))
```

**Note:-**

The maxsize argument tells lru_cache about how many recent return values to cache.

We can easily uncache the return values as well by using:

`foo.cache.clear()`

## 31.2 Python 2+

There are a couple of ways to achieve the same effect. You can create any type of caching mechanism. It entirely depends upon your needs. Here is a generic cache:

```
from functools import wraps

def memoize(function):
    memo = {}
    @wraps(function)
    def wrapper(*args):
        try:
            return memo[args]
        except KeyError:
            rv = function(*args)
            memo[args] = rv
            return rv
    return wrapper

@memoize
def fibonacci(n):
```

```
    if n < 2: return n
    return fibonacci(n - 1) + fibonacci(n - 2)

fibonacci(25)
```

**Note:-**

Note: memoize won't cache unhashable types (dict, lists, etc…) but only the immutable types. Keep that in mind when using it.

# 32 Regular Expressions

This section on regular expressions (regex) in Python will not provide a guide on how to use regex in general. Instead, it will specifically focus on explaining how regex is utilized within the context of Python.

Here's a brief explanation of how to use regular expressions (regex) in Python:

1.) **Import the re Module:**

Before using regex in Python, you need to import the re module, which provides functions and methods for working with regular expressions.

You can import it using the following statement:

**Example:**

```
import re
```

2.) **Create a Regex Pattern:**

In Python, a regex pattern is represented as a string. Define the pattern that you want to search for or match in your input text.

**Example:**

```
pattern = r"\d+"
```

3.) **Match the Pattern:**

There are several methods available in the re module to match patterns against input strings. Some commonly used methods include:

- **re.search(pattern, string, flags=0)**: Searches the string for a match to the pattern. Returns a match object if a match is found, or None otherwise.
- **re.match(pattern, string, flags=0)**: Matches the pattern against the beginning of the string. Returns a match object if a match is found, or None otherwise.
- **re.findall(pattern, string, flags=0)**: Returns all non-overlapping matches of the pattern in the string as a list of strings.
- **re.finditer(pattern, string, flags=0)**: Returns an iterator yielding match objects for all non-overlapping matches of the pattern in the string.
- **re.sub(pattern, repl, string, count=0, flags=0)**: used to replace match with some provided repl
- ***matchobject*.group()**: called on match objects to return only the match

4.) **Perform Operations on Matches:**

Once you have a match object, you can perform various operations on it. Some common operations include:

- Accessing the matched substring using the .group() method.
- Extracting specific parts of the match using capturing groups () in the pattern.
- Replacing matches in the input string using the re.sub(pattern, replacement, string) function.

5.) **Use Regex Modifiers and Flags (optional):**

If needed, you can use regex modifiers and flags to modify the matching behavior. For example, using the re.IGNORECASE flag for case-insensitive matching.

**Example: Find all words from string that are only 4 letters long**

```
#!/usr/bin/python3
import re

def findFourLetterWords(text=""):
    if text:
        pattern = r"\b\w{4}(?=\s)"
        return re.findall(pattern, text)
    else:
        print("Required string must not be empty")

def main():
    """ MAIN """

    text = "my name is nate and i like apples"

    result = findFourLetterWords(text)
    print(result)

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

# 33 Writing Tests

For our purposes, we will cover the two main types of tests:

- An **integration test** checks that components in your application operate with each other.
- A **unit test** checks a small component in your application.

You can write both integration tests and unit tests in Python. To write a unit test for the built-in function sum(), you would check the output of sum() against a known output.

For example, here's how you check that the sum() of the numbers (1, 2, 3) equals 6:

```
assert sum([2,3,1]) == 6 "should be 6"
```

This will not output anything on the REPL because the values are correct.

Instead of testing on the REPL, you'll want to put this into a new Python file called test_sum.py and execute it again:

```
def test_sum():
    assert sum([1, 2, 3]) == 6, "Should be 6"

if __name__ == "__main__":
    test_sum()
    print("Everything passed")
```

## 33.1 Choosing a test runner

There are many test runners available for Python. The one built into the Python standard library is called unittest. In this tutorial, you will be using unittest test cases and the unittest test runner. The principles of unittest are easily portable to other frameworks. The three most popular test runners are:

- unittest
- nose or nose2
- pytest

## 33.2 Unit Test

To convert the earlier example to a unittest test case, you would have to:

1. Import unittest from the standard library
2. Create a class called TestSum that inherits from the TestCase class
3. Convert the test functions into methods by adding self as the first argument
4. Change the assertions to use the self.assertEqual() method on the TestCase class
5. Change the command-line entry point to call unittest.main()

```
import unittest

class TestSum(unittest.TestCase):

    def test_sum(self):
        self.assertEqual(sum([1, 2, 3]), 6, "Should be 6")

    def test_sum_tuple(self):
        self.assertEqual(sum((1, 2, 2)), 6, "Should be 6")

if __name__ == '__main__':
    unittest.main()
```

Here is a table of all of the unittest methods that can be used by inheriting from unittest.TestCase

| Method | Equivalent to |
|---|---|
| .assertEqual(a, b) | a == b |
| .assertTrue(x) | bool(x) is True |
| .assertFalse(x) | bool(x) is False |
| .assertIs(a, b) | a is b |
| .assertIsNone(x) | x is None |
| .assertIn(a, b) | a in b |
| .assertIsInstance(a, b) | isinstance(a, b) |

## 33.3 Writing Your First Test

*main.py*

```
#!/usr/bin/python3
import testermane

def findsum(args):
    args = [x + 1 for x in args]
    return sum(args)


def main() -> int:
    """ MAIN """


    return 0


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

*testermane.py*

```
#!/usr/bin/python3
import unittest
import main

class TestSum(unittest.TestCase):
    def test1(self):
        testcase = (0,1,2)
        result = main.findsum(testcase)
        self.assertEqual(result,6)

    def test2(self):
        testcase = (3,0,0)
        result = main.findsum(testcase)
        self.assertEqual(result,6)

if __name__ == '__main__':
    unittest.main()
```

# 34 Functional Programming