Pythos

A CodeTime Odyssey

By R-m1n

Contents

Variables	1
Comments	2
Receiving Input	3
Strings	4
Arithmetic Operators	6
Bitwise Operators	8
If Statements	10
While Loop	11
Functions	12
Data Structures	13
Exceptions	28
Classes	30
Modules	51
Python Standard Library	53
Python Package Index	67

Sources

- codewithmosh
- geeksforgeeks
- w3schools
- programiz
- makeuseof
- heroku
- wikipedia

Most of the contents of this handbook are based on the teachings of everyone's favorite instructor **Mosh Hamedani**. Please check out his website https://codewithmosh.com/ and enjoy his wonderful courses on the topics of Software Engineering and Computer Science.

Variables

We use variables to temporarily store data in memory.

A Few Examples of Data-Types:

```
price = 10
rating = 4.9
name = 'Pythos'
is_published = True
```

- price is an integer (a whole number without a decimal point)
- rating is a float (a number with a decimal point)
- name is a string (a sequence of characters)
- is_published is a bool (boolean values can be True or False)

Comments

Comments can be used to add notes to our code. Good comments explain the hows and whys, not what the code does. That should be reflected in the code itself. Use comments to add reminders to yourself or other developers, or also explain your assumptions and the reasons you've written code in a certain way.

```
# This is a comment and it won't get executed.
```

Receiving Input

We can receive input from the user by calling the **input()** function.

```
birth_year = int(input('Birth year: '))
```

The `input()` function always returns data as a string. So, we're converting the result into an integer by calling the built-in `int()` function.

Strings

We can define strings using single (' ') or double (" ") quotes. To define a multi-line string, we surround our string with tripe quotes (""" """). We can get individual characters in a string using square brackets [ ].

```
course = 'Python'

course[0]  # returns the first character

course[1]  # returns the second character

course[-1] # returns the first character from the end
```

We can slice a string using a similar notation:

```
course[1:5]
```

The above expression returns all the characters starting from the index position of 1 to 5 (but excluding 5). The result will be **ytho** If we leave out the start index, 0 will be assumed. If we leave out the end index, the length of the string will be assumed.

We can use formatted strings to dynamically insert values into our strings:

```
name = 'Guido'
message = f'Hi, my name is {name}'

message.upper()
# to convert to uppercase

message.lower()
# to convert to lowercase

message.title()
# to capitalize the first letter of every word

message.find('p')
# returns the index of the first occurrence of p (or -1 if
  not found)

message.replace('p', 'q')

message.strip()
# remove white spaces from both sides

message.lstrip()

message.rstrip()
```

To check if a string contains a character (or a sequence of characters), we use the `in` operator, which returns a Boolean value:

```
contains = 'Python' in course
```

Arithmetic Operators

```
+

-

*

/  # returns a float

// # returns an int

%  # returns the remainder of division

** # exponentiation
```

Augmented Assignment Operator:

```
x  =  x + 10        ->        x  +=  10
```

Operator Precedence:

1. Parenthesis
2. Exponentiation
3. Multiplication / Division
4. Addition / Subtraction

Number Types:

```
x = 1       ∈ Z # integers

y = 1.1     ∈ R # real numbers (floating-point numbers)

z = 1 + 2j  ∈ C # complex numbers
```

Type Conversion:

```
str(arg)    # converts arg to a string

int(arg)    # converts arg to an integer

float(arg)  # converts arg to a float

bool(arg)   # converts arg to a boolean

type(arg)   # returns the type of arg
```

Boolean Values:

False Values:

`False / None / 0 / "" / () / [] / {}`

True Values:

Everything other than False values.

Bitwise Operators

Bitwise operators are used to perform bitwise calculations on integers. The integers are first converted into binary and then operations are performed bit by bit, hence the name bitwise operators. The result is in decimal format.

Bitwise AND Operator: Returns 1 if both of the bitsare 1 else 0.

```
a = 10 =  1010 (Binary)
b =  4 =  0100 (Binary)

a & b = 1010
         &
        0100
      = 0000
      = 0 (Decimal)
```

Bitwise OR Operator: Returns 1 if at least one ofthe bits is 1 else 0.

```
a = 10 = 1010 (Binary)
b =  4 = 0100 (Binary)

a | b = 1010
         |
        0100
      = 1110
      = 14 (Decimal)
```

Bitwise NOT Operator: Returns one's complement of the number.

```
a = 10 = 1010 (Binary)

~a =  ~1010
   = -(1010 + 1)
   = -(1011)
   = -11 (Decimal)
```

Bitwise XOR Operator: Returns 1 if the bits are opposite else 0.

```
a = 10 = 1010 (Binary)
b =  4 = 0100 (Binary)

a ^ b = 1010
         ^
        0100
      = 1110
      = 14 (Decimal)
```

Python Implementation of Bitwise Operations:

```
a = 10
b = 4

print("a & b =", a & b)
# print bitwise AND operation

print("a | b =", a | b)
# print bitwise OR operation

print("~a =", ~a)
# print bitwise NOT operation

print("a ^ b =", a ^ b)
# print bitwise XOR operation


output:
      a & b = 0
      a | b = 14
      ~a = -11
      a ^ b = 14
```

If Statements

```
if is_hot:
    print('hot day')
elif is_cold:
    print('cold day')
else:
    print('beautiful day')
```

Ternary Operator:

```
var = val1 if condition else val2
```

Logical Operators:

```
and # produces a value of True if and only if both of its
      operands are True.

or  # produces a value of True if at least one of its
      operands are True.

not # negates the value of its operand.
```

Comparison Operators:

```
a >  b  # greater than
a >= b  # greater than or equal to
a <  b  # less than
a <= b  # less than or equal to
a == b  # equals
a != b  # not equals
```

While Loop

```
i = 1
while i < 5:
    print(i)
    i += 1
```

For Loop

```
for i in range(1, 5):
    print(i)
```

- `range(5)` generates 0, 1, 2, 3, 4
- `range(1, 5)` generates 1, 2, 3, 4
- `range(1, 5, 2)` generates 1, 3

note: the else clause after while/for loops executes if the loop finishes without breaking.

Functions

Functions are used to break up code into small reusable chunks. These chunks are easier to read, understand and maintain.

```
def greet_user(name):
    print(f'Hi {name}')

greet_user('John')
```

Parameters are placeholders for the data we can pass to functions.
Arguments are the actual values we pass.

There are Two Types of Arguments:

- **Positional Arguments:** their position (order) matters.

- **Keyword Arguments:** position doesn't matter - we prefix them with the parameter name.

`*args`: by putting `*` before a parameter, our function takes in a tuple of arguments.

`**args`: by putting `**` before a parameter, our function takes in a set of key-value arguments which then will be returned as a dictionary.

`note: increase readability of functions by using the keyword argument (name of the parameter e.g. number) before passing the argument to the function e.g. number=arg.`

Data Structures

Lists:

Lists are dynamically sized arrays, which are not necessarily homogeneous. They're mutable, meaning they can be altered even after their creation. Lists in Python are ordered and have a definite count. The elements in a list are indexed according to a definite sequence and the indexing of a list is done with 0 being the first index. Each element in the list has its definite place in the list, which allows duplication of elements in the list, with each element having its own distinct place and credibility.

```
list = list(seq)

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

numbers[0]
# returns the first item

numbers[1]
# returns the second item

numbers[-1]
# returns the first item from the end

numbers[::-1]
# returns the list in reverse order

numbers.append(6)
# adds 6 to the end

numbers.insert(0, 6)
# inserts 6 at index 0

numbers.remove(6)
# removes 6
```

```
numbers.pop()
# removes the last item

numbers.clear()
# removes all the item

numbers.index(8)
# returns the index of first occurrence of 8

numbers.sort()
# sorts the list

numbers.reverse()
# reverses the list

numbers.copy()
# returns a copy of the list
```

Lists can be concatenated by using `+`, or can be multiplied by using `*`.

Unpacking Lists:

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

first, second, third = numbers
# the number of variables that we have on the left side of
  the assignment operator should be equal to the number of
  items in the list.
```

If we only want to unpack some of the items in the list we can pack the rest of the items in a separate list:

```
first, second, *other = numbers

first, *other, last = numbers

*other, second_to_last, last = numbers
```

Enumerate Function:

This function returns an enumerate object that when iterated over, returns a tuple of all of the items, plus their corresponding index.

```
letters = ['a', 'b', 'c']

for index, letter in enumerate(letters):

    print(index, letter)

output:
      0     “a”
      1     “b”
      2     “c”
```

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

```
list.count(val)
string.count(val)
```

Sorting Items in a List:

Ascending order:

```
list.sort()
new_list = sorted(list)
```

Descending order:

```
list.sort(reverse=True)
new_list = sorted(list, reverse=True)
```

to sort more complex items you should pass a reference of a specific function to the keyword argument (`key=`) of a sorting function/method.

```
products = [('p1', 10), ('p2', 9), ('p3', 11)]

products.sort(key=lambda item : item[1])
# sorting by the price of the product.
```

A lambda function is **a small anonymous function**. A lambda function can take any number of arguments, but can only have one expression.

```
lambda parameters : expression
```

Map Function:

The map function works as an iterator to return a result after applying a function to every item of an ***`iterable`***. It is used when you want to apply a single transformation function to all the ***`iterable`*** elements. The ***`iterable`*** and function are passed as arguments to the map in Python.

```
prices = list(map(lambda item : items[1], products))

output:
      [10, 9, 11]
```

Filter Function:

The filter function filters the given sequence with the help of a function that tests each element in the sequence to be true or not.

```
filtered = list(filter(lambda item : items >= 10, products))

print(filtered)

output:
      [(“product1”, 10), (“product3”, 11)]
```

List Comprehension:

List comprehension is **an easy and compact syntax for creating a list from a string or another list**. It is a very concise way to create a new list by performing an operation on each item in the existing list. List comprehension is considerably faster than processing a list using the for loop.

For Mapping:

```
prices = [item[1] for item in products]
```

For Filtering:

```
prices = [item for item in products if item[1] >= 10]

# there are also other comprehensions for sets and
  dictionaries.
```

Zip Function:

The zip function takes iterables or containers and returns a single iterable object, having mapped values from all the containers. It is used to map the similar index of multiple containers so that they can be used just using a single entity.

```
string = 'abc'
list1 = [1, 2, 3]
list2 = [10, 20, 30]

print(list(zip(string, list1, list2)))

output:
      [('a', 1, 10), ('b', 2, 20), ('c', 3, 30)]

# logical operator 'not' can be used to determine whether a
  list is empty or not.
```

Stacks & Queues:

Stacks and queues are **simple data structures that allow us to store and retrieve data sequentially**. In a stack, the last item we enter is the first to come out (LIFO). In a queue, the first item we enter is the first come out (FIFO). To efficiently use the queue data structure in python we should use the deque class from collections module. `deque(seq)`

```
from collections import deque

queue = deque([])
queue.append(1)
queue.append(2)
queue.popleft()

output:
      deque([2])
```

Tuples:

A tuple is basically a read-only list. It can be used to contain a sequence of objects which cannot be modified (immutable).

```
var = (1, 2)
var = 1, 2
var = 1,
var = tuple(seq)

# tuples can be concatenated and multiplied.

# when we use tuple comprehension, we'll get a generator
  object.
```

Generators:

Generators are iterables, that on each iteration generate a new value.

```
# especially useful when dealing with large data-set or
  potentially an infinite stream of data.

# because the generators don't store all the values inside
  memory we cannot get the total number of values.
```

Generator-Function: A generator-function is defined like a normal function, but whenever it needs to generate a value, it does so with the `yield` keyword rather than return. If the body of a `def` contains `yield`, the function automatically becomes a generator function.

Generator-Object: Generator functions return a generator object. Generator objects are used either by calling the next method on the generator object or using the generator object in a "`for in`" loop .

```python
# a simple generator for fibonacci numbers.

def fib(limit):
# initialize first two fibonacci numbers
    a, b = 0, 1
    # one by one yield next Fibonacci number
    while a < limit:
        a, b = b, a + b
        yield a
# create a generator object
x = fib(5)

# iterating over the generator object using next
print(x.next()) # __next__()
print(x.next())
print(x.next())
print(x.next())
print(x.next())

# iterating over the generator object using a for loop.
print('\nUsing for loop')
for i in fib(5):
    print(i)
```

output:

```
0
1
1
2
3

Using for loop

0
1
1
2
3
```

Advantages of Implementing Generators:

1. Easy to Implement

Generators can be implemented in a clear and concise way as compared to their iterator class counterpart.

2. Memory Efficient

A normal function to return a sequence will create the entire sequence in memory before returning the result.
Generator implementation of such sequences is memory friendly and is preferred since it only produces one item at a time.

3. Represent Infinite Stream

Generators are excellent mediums to represent an infinite stream of data. Infinite streams cannot be stored in memory, and since generators produce only one item at a time, they can represent an infinite stream of data.

4. Pipelining Generators

Suppose we have a generator that produces the numbers in the Fibonacci series. And we have another generator for squaring numbers.

If we want to find out the sum of squares of numbers in the Fibonacci series, we can do it in the following way by pipelining the output of generator functions together.

```
def square(nums):
    for num in nums:
        yield num**2

print(sum(square(fib(10))))
```

```
output:
      4895
```

Swapping Variables:

```
x, y = y, x
```

This is possible due to python's interpretation of the right-side as a tuple and with the assignment operator it unpacks the tuple into the variables on the left-side.

Pipeline:

In computing, a pipeline, also known as a data pipeline, is a set of data processing elements connected in series, where the output of one element is the input of the next one. The elements of a pipeline are often executed in parallel or in time-sliced fashion. Some amount of buffer storage is often inserted between elements.

Computer-Related Pipelines Include:

- **Instruction pipelines**, such as the classic RISC pipeline, which are used in central processing units (CPUs) and other microprocessors to allow overlapping execution of multiple instructions with the same circuitry. The circuitry is usually divided up into stages and each stage processes a specific part of one instruction at a time, passing the partial results to the next stage. Examples of stages are instruction decode, arithmetic/logic and register fetch. They are related to the technologies of superscalar execution, operand forwarding, speculative execution and out-of-order execution.

- **Graphics pipelines**, found in most graphics processing units (GPUs), which consist of multiple arithmetic units, or complete CPUs, that implement the various stages of common rendering operations (perspective projection, window clipping, color and light calculation, rendering, etc.).

- **Software pipelines**, which consist of a sequence of computing processes (commands, program runs, tasks, threads, procedures, etc.), conceptually executed in parallel, with the output stream of one process being automatically fed as the input stream of the next one. The Unix system call pipe is a classic example of this concept.

- **HTTP pipelining**, the technique of issuing multiple HTTP requests through the same TCP connection, without waiting for the previous one to finish before issuing a new one.

Arrays:

An array is a collection of items stored at contiguous memory locations. The idea is to store multiple items of the same type together. This makes it easier to calculate the position of each element by simply adding an offset to a base value, i.e., the memory location of the first element of the array (generally denoted by the name of the array). The base value is index 0 and the difference between the two indexes is the offset.

```
from array import array

var = array(typecode, seq)

# arrays are most useful when dealing with large data-sets
  or when encountering performance problems.
```

note: to get the size of an object, getsizeof(*obj*) function from sys module can be used.

Sets:

A set is basically a collection without any duplicates.

```
var = set(seq)   or   var = {val1, val2, ...}
```

Sets are specially powerful because of the mathematical operations which is supported by them:

Union:	`set1 | set2`
Intersection:	`set1 & set2`
Difference:	`set1 - set2`
Symmetric Difference:	`set1 ^ set2`

```
# sets are unordered collections of unique objects which
  means that they don't support indexing.
```

Dictionaries:

Dictionary is an unordered collection of key value pairs.

```
var = dict()
var = dict(seq)
var = {k1:v1, k2:v2}

.get(key, default_val)
# returns the value for the given key, or a default value if
  the value isn't in the dictionary.

.clear()
# removes all items from the dictionary.

.items()
# return a list with all dictionary keys with values.

.keys()
# returns a view object that displays a list of all the keys
  in the dictionary in order of insertion.

.values()
# returns a list of all the values available in a given
  dictionary.

.update(dict)
# updates the dictionary with the elements from another
  dictionary.

dict.fromkeys(seq, val)
# creates a dictionary from the given sequence, initialized
  with a value that will be assigned to the generated keys.
```

Unpacking Operators:

We use two operators * (for lists/tuples) and ** (for dictionaries).

```
def f(a, b, c, d):
    print(a, b, c, d)

my_list = [1, 2, 3, 4] # driver code

# unpacking list into four arguments
f(*my_list)
```

```
output:
      (1, 2, 3, 4)
```

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

d = {'a':2, 'b':4, 'c':10}

# a call with unpacking dictionary operator
f(**d)
```

```
output:
      2 4 10
```

```
# for cleaner printing, use pprint function from pprint
  module.
```

Exceptions

An exception is a kind of error that terminates the execution of our program.

Handling Exceptions:

```
try:
    Body

except ErrorType:
# or alternatively except ErrorType as et
    Body

else:
# /optional/ executes if no exceptions were raised.
    Body

finally:
# will always be executed regardless of having an exception.
    Body


# finally clause is mostly used to release external
  resources.
```

The With Statement:

The with statement is used to automatically release external resources.

```
with obj as var:
    Body

# automatic release only works with the objects that have
  .__exit__() method implemented.

# with statement can work with multiple objects at the same
  time.
```

note: by using timeit function from timeit module you can get the execution time of your code.

```
timeit(Code, number=int)

# Code is passed as a string of the code you want to
  evaluate.

# number keyword-argument determines the number of times the
  code will be executed.
```

Raising Exceptions:

```
raise ErrorType('MESSAGE')

# argument is optional.

# raising exceptions are costly and are advised to be
  preferably avoided.
```

note: pass is a null statement. which is a statement that doesn’t do anything.

Classes

A class is a blueprint for creating new objects, and an object is an instance of a class. Every time we create a new instance, that instance follows the structure we define in the class.

```
# use isinstance function to find out if an object is an
  instance of a specific class.

isinstance(obj, cls)
```

We use classes to define new types.

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

    def move(self):
        print('move')

point1 = Point(10, 5)
point2 = Point(2, 4)

# self is a reference to the current object.
```

Constructors:

Constructors are generally used for instantiating an object. The task of constructors is to initialize (assign values) to the data members of the class when an object of the class is created. The `__init__` magic method is called the constructor and is always called when an object is created.

```
def __init__(self):
    # body of the constructor
```

Default Constructor:

The default constructor is a simple constructor which doesn't take any arguments. It has only one argument which is a reference to the instance being constructed.

```
class Chanter:
    # default constructor
    def __init__(self):
        self.battle_cry = 'long live the BDFL'

    def chant(self):
        print(self.battle_cry)

# creating an object of a class
obj = Chanter()

# calling the instance method
obj.chant()

output:
      long live the BDFL
```

Parameterized Constructor:

Constructor with parameters is known as a parameterized constructor. The parameterized constructor takes its first argument as a reference to the instance being constructed known as self and the rest of the arguments are provided by the programmer.

```
class Sum:
    first = 0
    second = 0
    answer = 0

    def __init__(self, f, s):
        self.first = f
        self.second = s

    def display(self):
        print("First number = " + str(self.first))
        print("Second number = " + str(self.second))
        print("Sum = " + str(self.answer))

    def calculate(self):
        self.answer = self.first + self.second

obj = Sum(1000, 2000)
obj.calculate()
obj.display()

output:
      First number = 1000
      Second number = 2000
      Sum = 3000
```

Attributes:

Attributes are basically variables that include data about an object.

Class vs. Instance Attributes:

Instance Attributes:

Are attributes that belong to the instance of a class, so every object of this class can have different values for this attribute.

```
# instance attributes
class Citizen:
    def __init__(self):
        self.name = 'John Doe'
        self.salary = 4000

    def show(self):
        print(self.name)
        print(self.salary)

citizen1 = Citizen()

print("Dictionary form :", vars(citizen1))
print(dir(citizen1))
```

output:

```
Dictionary form : {'salary':4000,'name': 'xyz'}
['__doc__', '__init__', '__module__', 'name', 'salary', 'show']
```

Class Attributes:

Are defined at class-level and are shared across all instances of the class.

```
class Counter:
    count = 0   # class attribute

    def increase(self):
        Counter.count += 1

obj1 = Counter()
obj1.increase() # calling increase() on an object

print(obj1.count)

obj2 = Counter()
obj2.increase() # calling increase() on another object

print(obj2.count)
print(Counter.count)

output:
      1
      2
      2
```

Class vs. Instance vs. Static Methods:

Class Methods:

The `@classmethod` decorator is a built-in function decorator which is an expression that gets evaluated after your function is defined. The result of that evaluation shadows your function definition.
A class method receives the class as the implicit first argument, just like an instance method receives the instance.

```
class ClassExample():

    @classmethod
    def f(cls, arg1, arg2, ...):
       Body
```

Instance Methods:

An instance method can access and even modify the value of attributes of an instance. It has one default parameter, self.

```
class Shape:

    # calling constructor
    def __init__(self, edge, color):
        self.edge = edge
        self.color = color

    # instance method
    def edges(self):
        return self.edge

    # instance method
    def modifyEdges(self, newedge):
        self.edge = newedge
```

```
# driver code
circle = shape(0, 'red')
square = shape(4, 'blue')

# calling instance method
print("No. of edges for circle: " + str(circle.Edges()))

# calling instance method
square.modifyEdges(6)

print("No. of edges for square: " + str(square.Edges()))


output:
        No. of edges for circle: 0
        No. of edges for square: 6
```

Static Methods:

Are methods that are related to a class in some way, but don't need to access any class-specific data. instantiating an instance is optional, The `@staticmethod` decorator is used to declare a method is a static method. Static methods are great for utility functions, which perform a task in isolation. They cannot access class data, They should be completely self-contained, and only work with data passed in as arguments.

```
from datetime import date

class Person:

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

```
    @classmethod
    def fromBirthYear(cls, name, year):
        return cls(name, date.today().year - year)

    @staticmethod
    def isAdult(age):
        return age > 18

person1 = Person('mayank', 21)
person2 = Person.fromBirthYear('mayank', 1996)

print (person1.age)
print (person2.age)
print (Person.isAdult(22))

output:
        21
        25
        True
```

Magic Methods:

These are the methods that have two underscores at the beginning and end of their name and they are called automatically by the python interpreter depending on how we use our objects and classes.

`__new__(cls, other)`	To get called in an object's instantiation.
`__init__(self, other)`	To get called by the `__new__` method.
`__del__(self)`	Destructor method.
`__pos__(self)`	To get called for unary positive.
`__neg__(self)`	To get called for unary negative.
`__iadd__(self, other)`	To get called on augmented addition assignment.
`__isub__(self, other)`	To get called on augmented subtraction assignment.
`__iand__(self, other)`	To get called on bitwise AND with assignment.
`__ior__(self, other)`	To get called on bitwise OR with assignment.
`__int__(self)`	To get called by built-in `int()` method.
`__float__(self)`	To get called by built-in `float()` method.
`__str__(self)`	To get called by built-in `str()` method.
`__dir__(self)`	To get called by built-in `dir()` method.
`__sizeof__(self)`	To get called by built-in `sys.getsizeof()` function.
`__add__(self, other)`	To get called on addition operation.
`__sub__(self, other)`	To get called on subtraction operation.
`__lt__(self, other)`	To get called on comparison using < operator.
`__le__(self, other)`	To get called on comparison using <= operator.
`__eq__(self, other)`	To get called on comparison using == operator.
`__ne__(self, other)`	To get called on comparison using != operator.
`__ge__(self, other)`	To get called on comparison using >= operator.

Custom Containers:

```
class TagCloud:
    def __init__(self):
        self.tags = {}

    def add(self, tag):
        self.tags[tag.lower()] = \
             self.tags.get(tag.lower(), 0) + 1

    def __getitem__(self, tag):
        return self.tags.get(tag.lower(), 0)

    def __setitem__(self, tag, count):
        self.tags[tag.lower()] = count

    def __len__(self):
        return len(self.tags)

    def __iter__(self):
        return iter(self.tags)

cloud = TagCloud()
cloud["python"] = 10
cloud.add("pythos")
cloud.add("pythos")
cloud.add("pythos")

print(cloud.tags)
print(len(cloud))


output:
        {'python': 10, 'pythos': 3}
        2
```

Access Modifiers:

Modifiers in Python are only conventional and are not enforced by the interpreter.

Public Members:

Public members of a class are available to everyone. All members of a class are by default public in Python. These members can be accessed outside of the class, and their values can be modified.

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

Protected Members:

Protected members of a class can be accessed by other members within the class and are also available to their subclasses. Python has a unique convention to make a member protected: Add a prefix _ (single underscore).

```
class Modifiers:
    def __init__(self,name):
        # protected attribute
        self._protected_member = name
```

Private Members:

Private members of a class are only accessible within the class. In Python, a private member can be defined by using a prefix __ (double underscore).

```
class Modifiers:
    def __init__(self, name):
        # private attribute
        self.__private_member = name
```

Properties:

A property is an object that sits in front of an attribute and allows us to get/set the value of that attribute.

```
class Celsius:
    def __init__(self, temperature=0):
        self.temperature = temperature

    def to_fahrenheit(self):
        return (self.temperature * 1.8) + 32

    @property
    def temperature(self):
        return self._temperature

    @temperature.setter
    def temperature(self, value):
        self._temperature = value

human = Celsius(37)

print(human.temperature)
print(human.to_fahrenheit())
```

```
output:
        37
        98.60000000000001
```

Inheritance:

Inheritance is the capability of one class to derive or inherit the properties from another class. Benefits of Inheritance:

1. It represents real-world relationships well.
2. It provides reusability of a code. a technique to remove code duplication. Also, it allows us to add more features to a class without modifying it.
3. It is transitive in nature, which means that if class B inherits from another class A, then all the subclasses of B would automatically inherit from class A.

```
class Mammal:
    def walk(self):
        print('walk')

class Dog(Mammal):
    def bark(self):
        print('bark')

dog = Dog()
dog.walk()  # inherited from Mammal
dog.bark()  # defined in Dog
```

The Object Class:

All classes in python directly or indirectly inherit from the object class.

Use `issubclass()` function to find out if a class is a inherited from another class:

```
issubclass(cls, another_cls) # returns a Boolean value
```

Different Forms of Inheritance:

1. **Single Inheritance:** When a child class inherits from only one parent class, it is called single inheritance. We saw an example above.

2. **Multiple Inheritance:** When a child class inherits from multiple parent classes, it is called multiple inheritance.

3. **Multilevel Inheritance:** When we have a child and grandchild relationship.

4. **Hierarchical Inheritance:** More than one class is created from a single base, hierarchically.

5. **Hybrid Inheritance:** This form combines more than one form of inheritance. It is a blend of more than one type of inheritance.

note: It is advised to limit the levels of your inheritance to one or two, going beyond that only increases the complexity of the code and most likely would raise additional issues. Similarly, multi-level inheritance, if implemented improperly could introduce all sorts of issues and bugs into our program, it's best used when the parent classes are small, preferably abstract and have nothing in common.

Abstract Base Class:

Its purpose is to provide a common interface to its derivatives.

```
from abc import ABC, abstractmethod

class Polygon(ABC):
    @abstractmethod
    def sides(self):
        pass

class Triangle(Polygon):
    # implementing abstract method
    def sides(self):
        print("I have 3 sides")

class Pentagon(Polygon):
    def sides(self):
        print("I have 5 sides")

class Hexagon(Polygon):
    def sides(self):
        print("I have 6 sides")

r = Triangle()
r.sides()
p = Pentagon()
p.sides()
h = Hexagon()
h.sides()
```

note: Abstract classes cannot be instantiated. If a class is derived from an abstract base class it has to implement its abstract methods otherwise itself will be considered an abstract class.

Polymorphism:

The literal meaning of polymorphism is the condition of occurrence in different forms. It refers to the use of a single type entity (method, operator or object) to represent different types in different scenarios. In Python polymorphism can only be simulated using various techniques due the language's dynamically typed nature.

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

    def info(self):
        print(f"My name is {self.name}.")

    def make_sound(self):
        print("Meow")

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

    def info(self):
        print(f"My name is {self.name}.")

    def make_sound(self):
        print("Bark")

cat1 = Cat("Kitty")
dog1 = Dog("Fluffy")

for animal in (cat1, dog1):
    animal.make_sound()
    animal.info()
    animal.make_sound()
```

```
output:
        Meow
        I am a cat. My name is Kitty.
        Meow
        Bark
        I am a dog. My name is Fluffy.
        Bark
```

Polymorphism and Inheritance:

As we know, the child classes inherit methods and attributes from the parent class. We can redefine certain methods and attributes, which is called **Overriding**.
Polymorphism allows us to access these overridden methods and attributes that have the same name as the parent class.

```
from math import pi

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

    def area(self):
        pass

    def fact(self):
        return "I'm a two-dimensional shape."

    def __str__(self):
        return self.name

class Square(Shape):
    def __init__(self, length):
        super().__init__("Square")
        self.length = length
```

```
    def area(self):
        return self.length**2

    def fact(self):
        return "I’m friends with Pythagoras."

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

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

a = Square(4)
b = Circle(7)
print(b)
print(b.fact())
print(a.fact())
print(b.area())
```

output:

```
        Circle
        I’m a two-dimensional shape.
        I’m friends with Pythagoras.
        153.93804002589985
```

note: **Method Overloading,** which is a way to create multiple methods with the same name but different arguments, is not possible in Python.

Duck Typing:

"If it looks like a duck and quacks like a duck, it's a duck"

because python is a dynamically typed language we can achieve the same result as the previous example as follows:

```
class Specialstring:
    def __len__(self):
        return 21

if __name__ == "__main__":
    string = Specialstring()
    print(len(string))
```

```
output:
        21
```

note: because python supports duck typing we could achieve polymorphic behavior without using an abstract base class, having said that using an abstract base class is good practice because it enforces a uniform interface across all of its derivatives.

```
# id(obj) function returns the memory address of where the
  data is stored.
```

Extending Built-in Types:

```
class Text(str):
    def duplicate(self):
        return self + self

class TrackableList(list):
    def append(self, obj):
        print("Append called")
        super().append(obj)

l = TrackableList()
l.append("1")
print(l)
```

```
output:
        Append called
        ['1']
```

Data Classes:

For classes that only store/contain data we should use “namedtuples” instead of defining a class with attributes and no methods.

```
from collections import namedtuple

Point = namedtuple("Point", ["x", "y"])
p1 = Point(x=1, y=2)
print(p1.x)
p2 = Point(x=1, y=2)
print(p1 == p2)
```

```
output:
        1
        True
```

Decorators:

The decorators are used to modify the behavior of a function or a class. In decorators, functions are taken as the argument into another function and then called inside a wrapper function. In simple words, they are functions which modify the behavior of other functions.
Decorators have to precede a function or class declaration. They start with the @ sign. It's possible to combine multiple decorators, write your own, and apply them to classes as well.

```
class DecoratorExample:
        def __init__(self):
                print("Hello, World!")

        @staticmethod
        def example_function():
                print("I'm a decorated function!")

de = DecoratorExample()
de.example_function()
```

```
output:
        Hello, World!
        I'm a decorated function!
```

Modules

A module is a file/package that contains highly related python objects.

Compiled Python Files:

Every imported file will only be loaded once, and then will be cached into the memory as a python byte-code.

Module Search path:

When python sees an import statement it will search all the directories inside `sys.path` to find the module we want to import.

```
import sys

print(sys.path)
```

Importing all the objects in a modules:

```
import module *
```

Importing some the objects in a modules:

```
from module import object, ...
```

note: Importing all of the objects of a module by using * is bad practice, because in that module you could have several different functions or variables and if you blindly import them into the current file some of those objects may overwrite the objects with the same name in the current file.

Packages:

A package is a directory containing __init__.py . It can contain one or more modules.

```
# importing the entire sales module
from ecommerce import sales
sales.calc_shipping()

# importing one function in the sales module
from ecommerce.sales import calc_shipping
calc_shipping()
```

note: A package is a map to a directory, a module is a map to a file.

The dir Function:

With this function we can get the list of attributes and methods of an object.

```
dir(obj)
```

Executing Modules as Scripts:

All the statements inside a module will be executed the first time that the module is loaded. We can use this to write our initialization code for our packages.

note: In Python, strings prefixed with r or R, such as r'...' and R"...", are called raw strings and treat backslashes \ as literal characters. Raw strings are useful when handling strings that use a lot of backslashes, such as Windows paths and regular expression patterns.

Python Standard Library

Python comes with a huge library of modules for performing common tasks such as working with files, sending emails, working with date/time, generating random values, etc.

Paths:

pathlib module in Python provides various classes representing file system paths with semantics appropriate for different operating systems. This module comes under Python's standard utility modules.
Path classes in pathlib module are divided into pure paths and concrete paths. pure path provides only computational operations, but does not provide I/O operations, concrete path provides computational operations as it extends pure path, but it also provides I/O operations.

```
from pathlib import Path

path = Path(r'ecommerce\__init__.py')
print(path.exists())
print(path.is_file())
print(path.is_dir())
print(path.name)
print(path.stem)
print(path.suffix)
print(path.parent)
print(path.with_suffix('.txt'))
```

```
output:
        True
        True
        False
        __init__.py
        __init__
        .py
        ecommerce
        ecommerce\__init__.txt
```

Directories:

A directory or folder is a collection of files and subdirectories.

```
from pathlib import Path

path = Path('ecommerce')
path.exists()
path.mkdir() # create a new directory at this given path.
path.rmdir() # remove this directory, if empty.
path.rename('ecommerce2')
```

```
from pathlib import Path

path = Path('ecommerce')
paths = [p for p in path.iterdir() if p.is_dir()]
py_files = [p for p in path.glob('**/*.py')]

print(path)
print(py_files)
```

```
output:
        ecommerce
        [WindowsPath('ecommerce/__init__.py')]
```

Files:

The with statement:

Open a text file and read its contents:

```
with open('data.txt', 'r') as file:
    data = file.read()
```

To write data to a file, pass in w as an argument instead:

```
with open('data.txt', 'w') as file:
    data = 'some data to be written to the file'
    file.write(data)
```

Path Class:

```
from pathlib import Path

path = Path(r'ecommerce\__init__.py')
path.read_text(encoding=None, errors=None)
path.read_bytes()
path.write_text(data, encoding=None, errors=None,
                newline=None)
path.write_bytes(data)
```

Copying Files:

```
from pathlib import Path
import shutil

source = Path(r'ecommerce\__init__.py')
target = Path() / '__init__.py'
```

```
shutil.copy(source, target)
```

Zip Files:

Writing to a zip file:

```
from pathlib import Path
from zipfile import ZipFile

with ZipFile('files.zip', 'w') as zip:
    for path in Path('ecommerce').rglob('*.*'):
        zip.write(path)
```

Extracting a zip file:

```
from pathlib import Path
from zipfile import ZipFile

with ZipFile('files.zip') as zip:
    print(zip.namelist())
    info = zip.getinfo(r'ecommerce\__init__.py')
    print(info.file_size)
    print(info.compress_size)
    zip.extractall('extract')
```

CSV Files:

CSV: Comma Separated Value, it's like a simplified spreadsheet stored as a plain text file.

Writing to a CSV file:

```
import csv

with open('data.csv', 'w') as file:
    writer = csv.writer(file)
    writer.writerow(['transaction_id', 'product_id',
                     'price'])
    writer.writerow([1000, 1, 5])
    writer.writerow([1001, 2, 15])
```

Reading from a CSV file:

```
import csv

with open('data.csv') as file:
    reader = csv.reader(file)
    print(list(reader))
```

note: the reader object has an index reposition that is set to the reader object to a list that position goes to the end of the file so when we iterate over it we're already at the end of the file, in other words we cannot iterate the reader object twice.

JSON Files:

JSON: JavaScript Object Notation, is an open standard file format and data interchange format that uses human-readable text to store and transmit data objects consisting of attribute–value pairs and arrays (or other serializable values). It is a common data format with diverse uses in electronic data interchange, including that of web applications with servers. JSON is a language-independent data format.

Serialize Python:

```
import json
from pathlib import Path

movies = [
    {'id': 1, 'title': '12 Angry Men', 'year': 1957},
    {'id': 2, 'title': 'The Prestige', 'year': 2006}
]

data = json.dumps(movies)
Path('movies.json').write_text(data)
```

Parse JSON:

```
import json
from pathlib import Path

data = Path('movies.json').read_text()
movies = json.loads(data)
print(movies[0]['title'])
```

```
output:
        12 Angry Men
```

SQLite Database:

SQLite is a self-contained, high-reliability, embedded, full-featured, public-domain, SQL database engine. It is the most used database engine on the world wide web. Python has a library to access SQLite databases, called sqlite3, intended for working with this database which has been included with the Python package.

Connecting to the Database:

Connecting to the SQLite Database can be established using the `connect()` method, passing the name of the database to be accessed as a parameter. If that database does not exist, it'll be created.

```
import sqlite3

conn = sqlite3.connect('sql.db')
```

But what if you want to execute some queries after the connection is being made. For that, a cursor has to be created using the `cursor()` method on the connection instance, which will execute our SQL queries.

```
cursor = conn.cursor()
print('DB Init')
```

The SQL query to be executed can be written in the form of a string, and then executed by calling the `execute()` method on the cursor object. Then, the result can be fetched from the server by using the `fetchall()` method, which in this case, is the SQLite Version Number.

```
query = 'SQL query;'
cursor.execute(query)
result = cursor.fetchall()
print('SQLite Version is {}'.format(result))
```

Using SQLite3:

```
import sqlite3
import json
from pathlib import Path

movies = json.loads(Path("movies.json").read_text())

with sqlite3.connect("db.sqlite3") as conn:
    command = "INSERT INTO Movies VALUES(?, ?, ?)"
    for movie in movies:
        conn.execute(command, tuple(movie.values()))
    conn.commit()

# use sqlite browser to create tables for your database.
```

```
import sqlite3

with sqlite3.connect("db.sqlite3") as conn:
    command = "SELECT * FROM Movies"
    cursor = conn.execute(command)
    movies = cursor.fetchall()
    print(movies)
```

TimeStamps:

`time()` function from the time module returns the elapsed seconds since the beginning of time, which is operating system specific.

```
import time

def send_emails():
    for i in range(10000):
        pass

start = time.time()
send_emails()
end = time.time()
duration = end - start
print(duration)
```

DateTime:

```
from datetime import datetime
import time

dt1 = datetime(1984, 6, 6)
dt2 = datetime.now()
dt3 = datetime.strptime("1984/06/06", "%Y/%m/%d")
dt4 = datetime.fromtimestamp(time.time())

print(f"{dt3.year}/{dt3.month}/{dt3.day}")
print(dt4.strftime("%Y/%m"))
print(dt2 > dt1)
```

```
output:
        1984/6/6
        2022/04
        True
```

MIME: Multipurpose Internet Mail Extensions
TLS: Transport Layer Security

TimeDelta:

`timedelta()` function is present under the datetime library which is generally used for calculating differences in dates and also can be used for date manipulations in Python.

```
from datetime import datetime, timedelta

dt1 = datetime(1903, 12, 28) + timedelta(days=1,
                                         seconds=1000)
print(dt1)
dt2 = datetime.now()

duration = dt2 - dt1
print(duration)
print("days", duration.days)
print("total_seconds", duration.total_seconds())

output:
        1903-12-29 00:16:40
        43196 days, 0:36:06.151938
        days 43196
        total_seconds 3732136566.151938
```

Generating Random Values:

```
import random
import string

print(random.random())
print(random.randint(1, 10))
```

```
print(random.choice([1, 2, 3, 4]))
print(random.choices([1, 2, 3, 4], k=2))
print("".join(random.choices(string.ascii_letters +
                             string.digits, k=4)))
numbers = [1, 2, 3, 4]
random.shuffle(numbers)
print(numbers)

output:
        0.4483913565709343
        5
        3
        [3, 4]
        mw2a
        [3, 1, 4, 2]
```

Opening The Browser:

```
import webbrowser

print('Deployment Completed')
webbrowser.open('http://google.com')
```

Sending Emails:

```
from email.mime.multipart import MIMEMultipart
from email.mime.text import MIMEText
import smtplib

message = MIMEMultipart()
message["from"] = "Solaire"
message["to"] = "test@pythos.com"
message["subject"] = "Praise The Sun"
message.attach(MIMEText("Body"))
```

```
with smtplib.SMTP(host="smpt.gamil.com", port=587) as smtp:
    smtp.ehlo()
    smtp.starttls()
    smtp.login("test@pythos.com", "QXN0b3Jh")
    smtp.send_message(message)
```

Command Line Arguments:

The arguments that are given after the name of the program in the command line shell of the operating system are known as Command Line Arguments.
The sys module provides functions and variables used to manipulate different parts of the Python runtime environment. This module provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter.

```
import sys

tap = len(sys.argv)
print("Total arguments passed:", tap)
print("\nName of Python script:", sys.argv[0])
print("\nArguments passed:", end=" ")
for i in range(1, tap):
    print(sys.argv[i], end=" ")

sum = 0
for i in range(1, tap):
    sum += int(sys.argv[i])

print("\nResult:", sum)
```

```
cmd:   python add.py 2 3 5 6
       Total arguments passed: 5
       Name of Python script: add.py
       Arguments passed: 2 3 5 6
       Result: 16
```

Running External Programs:

This is particularly useful as part of an automation script, for example, we want our program to run the `dir` command, and capture the output. this command, lists the files and directories, in the current directory.
Or perhaps we want our script to run another script by using `python` command. To achieve this we should use the subprocess module, which consists of multiple helper methods namely:

```
.call() / .check_call() / .check_output()
```

To create an instance of Popen class. However these methods are considered legacy, there is a better and a newer approach by using the `.run()` method.

```
subprocess.run(*args, stdin=None, input=None, stdout=None,
               stderr=None, capture_output=False,
               shell=False, cwd=None, timeout=None,
               check=False, encoding=None, errors=None,
               text=None, env=None,
               universal_newlines=None,
               **other_popen_kwargs)
```

The return value of the `.run()` method is an instance of the `CompletedProcess` class.

```
# a process is an instance of a running program.
```

`subprocess.CompletedProcess` Class Attributes:

```
completed = subprocess.run(['dir', '/A'])
completed.args
# returns a list of passed arguments

completed.returncode
# returns 0 if success else error
```

```
completed.stderr
# returns a bytes sequence if captured else None

completed.stout
# returns the output if captured else None
```

To capture the output we should pass the keyword-argument `capture_output=True` to the `.run()` method.

```
# if we captured the output, it will not be printed on the
  terminal, instead it will be available at stout attribute
  of our instance of the CompletedProcess class.
```

The type of data stored in the `stout` attribute is a bytes sequence by default, in order to change it into a regular string we should pass another keyword-argument `text=True` to the `.run()` method.

note: when executing another script as a child process, it'll be in another completely different memory space which is different from importing that script in the current script, so they will not share the same variables.

By passing the keyword-argument `check=True` to `.run()` method, if there's an error it'll automatically raise an exception.

Python Package Index

PyPI:

The Python Package Index, also known as the Cheese Shop, is the official third-party software repository for Python. PyPI is run by the Python Software Foundation. Some package managers, including pip, use PyPI as the default source for packages and their dependencies. PyPI primarily hosts Python packages in the form of archives called sdists (source distributions) or precompiled "wheels". PyPI as an index allows users to search for packages by keywords or by filters against their metadata, such as free software license or compatibility with POSIX. A single entry on PyPI is able to store, aside from just a package and its metadata, previous releases of the package, precompiled wheels (e.g. containing DLLs on Windows), as well as different forms for different operating systems and Python versions.

pip:

pip is a package-management system used to install and manage software packages. It connects to an online repository of public packages, namely PyPI. pip can also be configured to connect to other package repositories (local or remote), provided that they comply with Python Enhancement Proposal 503.

```
pip install packageName

pip uninstall packageName
```

pip has a feature to manage full lists of packages and corresponding version numbers, possible through a "requirements" file. This permits the efficient re-creation of an entire group of packages in a separate environment or virtual environment. This can be achieved with a properly formatted file and the following command.

```
pip install -r requirements.txt
```

Virtual Environment:

Suppose we have a project in which we need to use an earlier version of a certain package, for this purpose we need to create an isolated, virtual environment for each application and install these dependencies in that virtual environment.

Creating a Virtual Environment and Pointing it to a Directory env:

```
python -m venv env
```

Activating the Virtual Environment:

```
env\bin\activate.bat
```

Installing a Package with Optionally Specified Version:

```
(env) pip install packageName==2.9.*
```

Exiting the Virtual Environment:

```
(env) deactive
```

pipenv:

pipenv is a tool that combines pip and virtual environments into a single tool chain. so we don't have to use them separately. It's basically a dependency manager for python projects.

Installation:

```
pip install pipenv
```

Using pipenv to Install a Package Inside a Virtual Environment:

```
pipenv install packageName
```

Get the Path to the Virtual Environment Directory:

```
pipenv -venv
```

Activating the Virtual Environment for the Current Project:

```
pipenv shell
```

Deactivating the Virtual Environment:

```
exit
```

pipfile and pipfile.lock:

When we install a package using pipenv two files get automatically created, pipfile and pipfile.lock these files are used to keep track of the dependencies of our application and their version.

pipfile:

[[source]]

It specifies the address of the repository where the packages get downloaded from.

[dev-package]

These are packages that we use as part of our development such as the package that we use for automated testing, they're only used during development, so they don't need to exist for our application to run.

[package]

These are packages that our application is dependent on.

[requires]

This specifies the version of python we need to run our application.

pipfile.lock:

This is a JSON file that lists the dependencies of our application and their exact version.

```
{
      "default" : "lists all the dependencies of our
      application and the packages that we installed and
      their dependencies, and their dependencies and so on
      , with their exact version."
}
```

With all the information stored in this file we can take our source code and put it in another machine like a production environment which in other words maximizes the portability of our application.

Installing all the Dependencies of our Application:

```
pipenv install
```

By using this command pipenv will look at the two files pipfile and pipfile.lock and will install all the dependencies of our application.

Installing all the Dependencies only from pipfile.lock File:

```
pipenv install --ignore-pipfile
```

This is used when we want the dependencies to have the exact version as the time of development.

Managing Dependencies:

There are some useful `pipev` commands for managing our application dependencies.

Getting the List of all the Installed Dependencies:

```
pipenv graph
```

Finding the Outdated Packages:

```
pipenv update --outdated
```

Updating all the Packages:

```
pipenv update
```

Updating a Specified Package:

```
pipenv update packageName
```

Publishing Packages:

To publish our own package first we need to create a pypi.org account.

Install Three Package Globally:

```
pip install setuptools wheel twine
```

Make a New Directory to Act as our Root Directory:

```
mkdir packageName
```

As a best practice we should make a high-level directory with the same name as our package to act as our source code directory, and also we could create another directory for unit testing named tests, and perhaps another one for our sample data named data.

We need to make sure to add `__init__.py` to our source code directory for python to see it as a package.

Now we can add a couple of modules to our source code directory e.g. module1.py, module2.py, … .

After we created our modules, we need to add three files to our root directory in order to publish this package to pypi.

1. **Setup.py :**

```
import setuptools
from pathlib import Path

setuptools.setup(name='packageName', version=1.0,
long_description=Path('README.md').read_text(),
package=setuptools.find_packages(exclude=['tests','data']))
```

2. **README.md :**

   A summary of our package to be shown at the homepage of our projects.

3. **LICENSE :**

   We can find a basic license template at choosealicense.com and copy it in this file. (e.g. GNU GPLv3)

Now have to Generate a Distribution Package:

```
python setup.py sdist bdist_wheel
```

This Command Generates Two Distribution Packages:

- Source distribution
- Build distribution

After this command two directories will automatically be created in our root directory, **dist** and **build**. dist directory contains two zip files, a **.whl** file, which is a build distribution and a **.tar.gz** file, which is a source distribution.

The final step is to upload them to pypi.org :

```
twine upload dist/*
```

Docstrings:

In Python there's a special format for documenting our code called docstrings, which is basically a string with triple-quotes that we add right after declaration of a function, a class or a variable. This is different from using comments, we should use comments to explain why we have done things in a certain way, not what/how something does/what it does, that should be reflected in the documentation string.
For public APIs it's practice to document the module itself as well as every object in it. (any classes, methods, standalone functions, …)

Docstring Conventions:

```
"""
One line description.

A more detailed explanation.
"""
```

```
"""
One line description.

A more detailed explanation.

Parameters
----------
parameter: type
      Summary

Return
------
type
"""
```

Pydoc:

In Python there's a built-in utility called `pydoc` which we can use to easily see the documentation of a module.

```
pydoc moduleName
```

Writing the desired documentation to an html file:

```
pydoc -w moduleName
```

Loading the documentation of a given module as well as the documentation of Python standard library in a web server:

```
pydoc -p port
```