Functions

- A ***function*** is a block of code which only runs when it is called.

  You can pass data, known as *parameters*, into a function.

  A function can return data as a result.

```
def my_function():
  print("Hello from a function")
```

- Information can be passed into functions as *arguments*.

  Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma.

  A function must be called with the correct number of arguments.

- ```
  def my_function(fname):
    print("Mr. " + fname)

  my_function("Stark") ---> Mr. Stark
  ```

- From a function's perspective:

  A parameter is the variable listed inside the parentheses in the function definition.

  An argument is the value that is sent to the function when it is called.
  ```

Functions

- ```
  def myfunc():
    x = 300
    def myinnerfunc():
      print(x)
    myinnerfunc()
  myfunc()
  ```
- Python doesn’t allow for multiple definitions of the same function. In some languages, you can define a function multiple times, each time having a different signature. In Python, this functionality doesn’t exist; when you define a function, you’re assigning to a variable. And just as you can’t expect that x will simultaneously contain the values 5 and 7, you similarly can’t expect that a function will contain multiple implementations. The way that we get around this problem in Python is with flexible parameters. Between default values, variable numbers of arguments (*args), and keyword arguments (**kwargs), we can write functions that handle a variety of situations.
  ```

Function Arguments

- ## Arbitrary Arguments (**args*)

  If you do not know how many arguments that will be passed into your function, add a * before the parameter name in the function definition.

```
def my_function(*kids):
  print("The youngest child is " + kids[2])

my_function("Arthur", "Thomas", "Finn")
```

```
Output: The youngest child is Finn
```

- ## Keyword Arguments

You can also send arguments with the *key* = *value* syntax. This way the order of the arguments does not matter.

```
def my_function(child3, child2, child1):
  print("The youngest child is " + child3)

my_function(child1 = "Arthur", child2 = "Thomas", child3 = "Finn")
```

```
Output: The youngest child is Finn
```

Function Arguments

- ## Arbitrary Keyword Arguments (**kwargs*)

  If you do not know how many keyword arguments that will be passed into your function, add two asterisk: ** before the parameter name in the function definition.

```
def my_function(**kid):

  print("His last name is " + kid["lname"])

my_function(fname = "Thomas", lname = "Shelby")---> His last name is Shelby
```

- ## Default Parameter Value

  If we call the function without argument, it uses the default value.

```
def my_function(country = "India"):

 print("I am from " + country)

my_function("Norway") ---> I am from Norway

my_function() ---> I am from India
```

Function Arguments

- You can send any data types of argument to a function (string, number, list, dictionary etc.).
- To let a function return a value, use the return statement.

```
def my_function(x):
  return 5 * x
my_function(3) ---> 15
```

- function definitions cannot be empty, but if you for some reason have a function definition with no content, put in the pass statement to avoid getting an error.

```
def myfunction():
  pass
```

- You are allowed to use keyword arguments even if the function expects positional arguments.

```
def my_function(x):
  print(x)
my_function(x = 3)
```

Function Arguments

- You can specify that a function can have ONLY positional arguments. Add , / *after* the arguments.

```
def my_function(x, /):
  print(x)

my_function(3) # Now my_function(x=3) will show an Error.
```

- To specify that a function can have only keyword arguments, add *, *before* the arguments.

```
def my_function(*, x):
  print(x)

my_function(x = 3) # Now my_function(3) will show an Error.
```

-
```
def my_function(a, b, /, *, c, d):
  print(a + b + c + d)

my_function(5, 6, c = 7, d = 8)
```

Lambda

- A lambda function is a small anonymous function.
- A lambda function can take any number of arguments, but can only have one expression.
- `lambda` *`arguments`* `:` *`expression`*
  The *`expression`* is executed and the *result* is returned:
- ```
  x = lambda a : a + 10
  print(x(5))
  ```
- ```
  x = lambda a, b : a * b
  print(x(5, 6))
  ```
- Let's assume we have a function definition that takes one argument, and that argument will be multiplied with an unknown number:
  ```
  def myfunc(n):
    return lambda a : a * n
  ```
- We can use that function definition to make a function that always doubles the number you send in:
  ```
  def myfunc(n):
    return lambda a : a * n

  mydoubler = myfunc(2)
  print(mydoubler(11))
  ```

Lambda

- We can also use the same function definition to make a function that always *triples* the number you send in:

```
def myfunc(n):
  return lambda a : a * n

mytripler = myfunc(3)

print(mytripler(11))
```

- We can use the same function definition to make both functions, in the same program:

```
def myfunc(n):
  return lambda a : a * n

mydoubler = myfunc(2)

mytripler = myfunc(3)

print(mydoubler(11))

print(mytripler(11))
```

Exception Handling

- When an error occurs, or exception as we call it, Python will normally stop and generate an error message.

```
try:
  print(x)
except NameError:
  print("Variable x is not defined")
except:
  print("Something else went wrong")
```

- You can use the else keyword to define a block of code to be executed if no errors were raised.

```
try:
  print("Hello")
except:
  print("Something went wrong")
else:
  print("Nothing went wrong")
```

Exception Handling

- The finally block, if specified, will be executed regardless if the try block raises an error or not.

```
try:
  print(x)
except:
  print("Something went wrong")
finally:
  print("The 'try except' is finished")
```

- You can use the else keyword to define a block of code to be executed if no errors were raised.

```
try:
  f = open("demofile.txt")
  try:
    f.write("Lorum Ipsum")
  except:
    print("Something went wrong when writing to the file")
  finally:
    f.close()
except:
  print("Something went wrong when opening the file")
```

Exception Handling

- You can choose to throw an exception if a condition occurs.

  The raise keyword is used to raise an exception.

```
x = -1

if x < 0:

  raise Exception("Sorry, no numbers below zero")
```

- You can define what kind of error to raise.

```
x = "hello"

if not type(x) is int:

  raise TypeError("Only integers are allowed")
```

User Input

- ```
  username = input("Enter username:")

  print("Username is: " + username)
  ```

- By default the input given by user is treated as a *string*.
  ```

Modules

- Consider a module to be the same as a code library.

  A file containing a set of functions you want to include in your application.

- To create a module just save the code you want in a file with the file extension .py.

  Save this code in a file named mymodule.py.

```
def greeting(name):

  print("Hello, " + name)
```

- Now we can use the module we just created, by using the import statement.

```
import mymodule

mymodule.greeting("Jonathan")
```

- When using a function from a module, use the syntax: *module_name.function_name*.

Modules

- The module can contain functions, as already described, but also variables of all types (arrays, dictionaries, objects etc).

```
person1 = {
  "name": "John",
  "age": 36,
  "country": "Norway"
}
```

Save this code in the file mymodule.py.

```
import mymodule

a = mymodule.person1["age"]
```

Modules

- You can name the module file whatever you like, but it must have the file extension .py.
- You can create an alias when you import a module, by using the as keyword.
- Create an alias for mymodule called mx.

```
import mymodule as mx

a = mx.person1["age"]
```

- You can choose to import only parts from a module, by using the from keyword.
- The module named mymodule has one function and one dictionary.

```
from mymodule import person1

print (person1["age"])
```

File Handling

- The key function for working with files in Python is the open() function.
- The open() function takes two parameters; *filename*, and *mode*.
- There are four different methods (modes) for opening a file:
  - `"r"` - Read - Default value. Opens a file for reading, error if the file does not exist.
  - `"a"` - Append - Opens a file for appending, creates the file if it does not exist.
  - `"w"` - Write - Opens a file for writing, creates the file if it does not exist.
  - `"x"` - Create - Creates the specified file, returns an error if the file exists.
- In addition you can specify if the file should be handled as binary or text mode:
  - `"t"` - Text - Default value. Text mode
  - `"b"` - Binary - Binary mode (e.g. images)
- `f = open("demofile.txt", "rt")`

Read Files

- Assume we have the following file, located in the same folder as Python.
- The open() function returns a file object, which has a read() method for reading the content of the file:

```
f = open("demofile.txt", "r")
print(f.read())
```

- If the file is located in a different location, you will have to specify the file path, like this:

```
f = open("D:\\myfiles\welcome.txt", "r")
print(f.read())
```

- By default the read() method returns the whole text, but you can also specify how many characters you want to return:

```
f = open("demofile.txt", "r")
print(f.read(5))
```

- You can return one line by using the readline() method:

```
f = open("demofile.txt", "r")
print(f.readline())
```

- By looping through the lines of the file, you can read the whole file, line by line:

```
f = open("demofile.txt", "r")
for x in f:
  print(x)
```

- It is a good practice to always close the file when you are done with it.

```
f = open("demofile.txt", "r")
print(f.readline())
f.close()
```

* *You should always close your files. In some cases, due to buffering, changes made to a file may not show until you close the file.*

Create Files

- To create a new file in Python, use the open() method, with one of the following parameters:
  - "x" - Create - will create a file, returns an error if the file exists.
  - "a" - Append - will create a file if the specified file does not exists.
  - "w" - Write - will create a file if the specified file does not exists.
- Create a file called "myfile.txt":

```
f = open("myfile.txt", "x")
```

Result: a new empty file is created!

- Create a new file if it does not exist:

```
f = open("myfile.txt", "w")
```

Write Files

- To write to an existing file, you must add a parameter to the open() function:
  - "a" - Append - will append to the end of the file.
  - "w" - Write - will overwrite any existing content.
- Open the file "demofile2.txt" and append content to the file:

```
f = open("demofile2.txt", "a")
f.write("Now the file has more content!" )
f.close()
```

- Open the file "demofile3.txt" and overwrite the content:

```
f = open("demofile3.txt", "w")
f.write("Woops! I have deleted the content!" )
f.close()
```

* *The "w" method will overwrite the entire file.*

Delete Files

- To delete a file, you must import the OS module, and run its os.remove() function:

```
import os

os.remove("demofile.txt")
```

- To avoid getting an error, you might want to check if the file exists before you try to delete it:

```
import os

if os.path.exists("demofile.txt"):

  os.remove("demofile.txt")

else:

  print("The file does not exist")
```

- To delete an entire folder, use the os.rmdir() method:

```
import os

os.rmdir("myfolder")
```

* *You can only remove empty folders.*

Python Classes and Objects

- A Class is like an object constructor, or a "blueprint" for creating objects.
- To create a class, use the keyword class:

```
class MyClass:
  x = 5
```

- Now we can use the class named MyClass to create objects:

```
p1 = MyClass()
print(p1.x)
```

- The *__init__()* Function:
  - All classes have a function called __init__(), which is always executed when the class is being initiated.
  - Use the __init__() function to assign values to object properties, or other operations that are necessary to do when the object is being created.

Python Classes and Objects

```
class Person:

  def __init__(self, name, age):

    self.name = name

    self.age = age

p1 = Person("Luke", 36)

print(p1.name)

print(p1.age)
```

* *The __init__() function is called automatically every time the class is being used to create a new object.*

Python Classes and Objects

- The *__str__()* Function:
  - The __str__() function controls what should be returned when the class object is represented as a string.
  - If the __str__() function is not set, the string representation of the object is returned.

```
class Person:

  def __init__(self, name, age):

    self.name = name

     self.age = age

  def __str__(self):

    return f"{self.name}({self.age})"

p1 = Person("Luke", 36)

print(p1)
```

Object Methods

- Objects can also contain methods. Methods in objects are functions that belong to the object.

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age
  def myfunc(self):
    print("Hello, my name is " + self.name)
p1 = Person("Luke", 36)
p1.myfunc()
```

The *self* Parameter

- The *self* parameter is a reference to the current instance of the class, and is used to access variables that belong to the class.
- It does not have to be named *self*, you can call it whatever you like, but it has to be the first parameter of any function in the class:

```
class Person: # Use mysillyobject and abc instead of self

  def __init__(mysillyobject, name, age):

    mysillyobject.name = name

     mysillyobject.age = age

  def myfunc(abc):

    print("Hello, my name is " + abc.name)



p1 = Person("Luke", 36)

p1.myfunc()
```

Modify Object Properties

- Set the *age* of *p1* to 40:

```
p1.age = 40
```

- Delete the *age* property from the *p1* object:

```
del p1.age
```

- Delete the *p1* object:

```
del p1
```

- class definitions cannot be empty, but if you for some reason have a class definition with no content, put in the pass statement to avoid getting an error.

```
class Person:
  pass
```

Python Inheritance

- Inheritance allows us to define a class that inherits all the methods and properties from another class.
- **Parent class** is the class being inherited from, also called base class.
- **Child class** is the class that inherits from another class, also called derived class.
- Create a Parent class:

  Any class can be a parent class, so the syntax is the same as creating any other class.

```
class Person:

  def __init__(self, fname, lname):

    self.firstname = fname

     self.lastname = lname

  def printname(self):

     print(self.firstname, self.lastname)

#Use the Person class to create an object, and then execute the printname method:

x = Person("Obi-wan", "Kenobi")

x.printname()
```

Python Inheritance

- Create a Child class:

  To create a class that inherits the functionality from another class, send the parent class as a parameter when creating the child class:

```
class Student(Person):

  pass
```

  Create a class named Student, which will inherit the properties and methods from the Person class.

```
x = Student("Luke", "Skywalker")

x.printname()
```

- Add the __init__() function to the Student class:

```
class Student(Person):

  def __init__(self, fname, lname):

    #add properties etc.
```

The child's __init__() function overrides the inheritance of the parent's __init__() function.

Python Inheritance

- To keep the inheritance of the parent's __init__() function, add a call to the parent's __init__() function:

```
class Student(Person):

  def __init__(self, fname, lname):

     Person.__init__(self, fname, lname)
```

- Using the super() function:

```
class Student(Person):

  def __init__(self, fname, lname):

    super().__init__(fname, lname)
```

** By using the `super()` function, you do not have to use the name of the parent element, it will automatically inherit the methods and properties from its parent.*

Add Properties to the child class

- Add a property called graduationyear to the Student class:

```
class Student(Person):
  def __init__(self, fname, lname):
    super().__init__(fname, lname)
    self.graduationyear = 2025
```

- In the example, the year 2025 should be a variable, and passed into the Student class when creating student objects. To do so, add another parameter in the __init__() function.

```
class Student(Person):
  def __init__(self, fname, lname, year):
    super().__init__(fname, lname)
    self.graduationyear = year


x = Student("Luke", "Skywalker", 2025)
```

Add Methods

- Add a method called welcome to the Student class.

```
class Student(Person):

  def __init__(self, fname, lname, year):

    super().__init__(fname, lname)

    self.graduationyear = year

  def welcome(self):

    print("Welcome", self.firstname, self.lastname, "to the class
of", self.graduationyear)
```

Iterators

- An iterator is an object that can be iterated upon, meaning that you can traverse through all the values.
- In Python, an iterator is an object which implements the iterator protocol, which consist of the methods __iter__() and __next__().
- Lists, tuples, dictionaries, and sets are all iterable objects. They are iterable *containers* which you can get an iterator from. Strings are also iterable.
- All these objects have a iter() method which is used to get an iterator:
- ```
  mytuple = ("apple", "banana", "cherry")

  myit = iter(mytupl)
  ```
- *The for loop actually creates an iterator object and executes the next() method for each loop.*
  ```

Create an Iterator

- To create an object/class as an iterator you have to implement the methods __iter__() and __next__() to your object.
- The __iter__() method acts similar to __init__(), you can do operations (initializing etc.), but must always return the iterator object itself.
- The __next__() method also allows you to do operations, and must return the next item in the sequence.
-

```
class MyNumbers:

  def __iter__(self):

     self.a = 1

    return self

  def __next__(self):

    x = self.a

     self.a += 1

    return x

myclass = MyNumbers()

myiter = iter(myclass)

print(next(myiter))
```

StopIteration

- In the previous example above would continue forever if you had enough next() statements, or if it was used in a for loop.
- To prevent the iteration from going on forever, we can use the StopIteration statement.
- In the __next__() method, we can add a terminating condition to raise an error if the iteration is done a specified number of times:

```
class MyNumbers:

  def __iter__(self):

        self.a = 1

    return self

  def __next__(self):

    if self.a <= 20:

        x = self.a

      self.a += 1

      return x

    else:

      raise StopIteration

myclass = MyNumbers()

myiter = iter(myclass)

for x in myiter:

  print(x)
```

Problems on Strings and More

- Given a string *S* with the value `"s,pa,m"`, name two ways to extract the two characters in the middle.
- ***Pig Latin*** (*http://mng.bz/YrON*) is a common children's *secret* language in Englishspeaking countries. The rules for translating words from English into Pig Latin are quite simple: If the word begins with a vowel (`a, e, i, o,` or `u`), add "`way`" to the end of the word. So "`air`" becomes "`airway`" and "`eat`" becomes "`eatway`". If the word begins with any other letter, then we take the first letter, put it on the end of the word, and then add "`ay`". Thus, "`python`" becomes "`ythonpay`" and "`computer`" becomes "`omputercay`".
- Convert the following sentence "`this is a test`" to a Pig-Latin.
- Sort a given string: "`abcde`"
- Sort strings in a sentence: "`string sort with sentences`"
- Dynamic Typing.

Problems on Lists

- Find the Majority Element:

  You are given a list of integers. A majority element is an element that appears more than half the time in the list.

  Write a function *find_majority(nums)* that returns the majority element if it exists, otherwise return None.

```
nums = [3, 3, 4, 2, 4, 4, 2, 4, 4]
```

- Merging Overlapping Intervals:

  You are given a list of time intervals as *(start, end)* tuples. Your task is to merge overlapping intervals into a single interval whenever possible.

```
intervals1 = [(1, 3), (2, 6), (8, 10), (9, 11)]
intervals2 = [(1, 5), (6, 10), (11, 15)]
```

Problem on Tuples

- Task Scheduling (Find Overlapping Meetings):

  You are given a list of meeting time intervals in the form of *tuples (start_time, end_time)*. Your task is to determine if a person can attend all meetings without any overlaps.

```
meetings1 = [(1, 3), (2, 4), (5, 6)]
meetings2 = [(1, 2), (3, 4), (5, 6)]
```

Problem on Dictionaries

- Create a new constant dict, called MENU, representing the possible items you can order at a restaurant. The keys will be strings, and the values will be prices (i.e., integers). You should then write a function, restaurant, that asks the user to enter an order: If the user enters the name of a dish on the menu, the program prints the price and the running total. It then asks the user again for their order. If the user enters the name of a dish not on the menu, the program scolds the user (mildly). It then asks the user again for their order. If the user enters an empty string, the program stops prompting and prints the total amount. You can order same order multiple times. Inputs should be case-insensitive.

```
MENU = {'sandwich': 10, 'tea': 7, 'salad': 9}
```

Problem on Dictionaries

- Write a function, *get_rainfall*, that tracks rainfall in a number of cities. Users of your program will enter the name of a city; if the city name is blank, then the function prints a report (*city_name: rainfall amount*) before exiting. If the city name isn't blank, then the program should also ask the user how much rain has fallen in that city (typically measured in millimeters). After the user enters the quantity of rain, the program again asks them for a city name, rainfall amount, and so on—until the user presses Enter instead of typing the name of a city. When the user enters a blank city name, the program exits—but first, it reports how much total rainfall there was in each city. The order in which the cities appear is not important, and the cities aren't known to the program in advance.

Problems on Function

- Write a function, *myxml*, that allows you to create simple XML output. The output from the function will always be a string. The function can be invoked in a number of ways, as shown in the table.

Call	Return Value
`myxml('foo')`	`<foo></foo>`
`myxml('foo', 'bar')`	`<foo>bar</foo>`
`myxml('foo', 'bar', a=1, b=2, c=3)`	`<foo a="1" b="2" c="3">bar</foo>`

- `Input: 'tagname', 'hello', a=1, b=2, c=3)`

Problems on Function

- Write a function (*calc*) that expects a single argument—a string containing a simple math expression in prefix notation—with an operator and two numbers. Your program will parse the input and produce the appropriate output. For our purposes, it's enough to handle the six basic arithmetic operations in Python: addition (+), subtraction (-), multiplication (*), division (/), modulus (%), and exponentiation (**). The normal Python math rules should work, such that division always results in a floating-point number. We'll assume, for our purposes, that the argument will only contain one of our six operators and two valid numbers.

  Input: + 2 3

- Write a program to evaluate any general prefix expression.

Miscellaneous

In a **call center**, customers wait in line to be assisted by a support agent. The support agent serves customers in the order they arrive, which follows the **First Come, First Served (FCFS)** principle.

Your task is to simulate a call center service using Python classes. Implement the following requirements:

Requirements:

1. **Customer Queue**:
    - When a customer calls in, they are added to a **waiting queue**.
    - The customers are served in the **order they arrive** (FIFO - First In, First Out).
2. **Support Agent**:
    - There is a single **support agent** available at any given time.
    - The agent can only serve one customer at a time.
    - The agent's availability is managed and is initially available when the program starts.
3. **Queue Management**:
    - Implement a **queue** to hold customers who are waiting to be served.
    - The program should be able to **add customers** to the queue when they arrive.
    - If there are **no customers** in the queue, the program should print **"empty queue"**.
4. **Serving Customers**:
    - Implement a ***serve_customer()*** function that:
        - Dequeues the customer from the front of the queue and serves them.
        - If the queue is empty, the program should print **"no customer"**.
    - After serving a customer, the agent becomes available again and is ready to serve the next customer.
5. **Queue Visualization**:
    - Implement a function ***show_queue()*** that displays the current customers in the queue.
    - If the queue is empty, it should print **"empty queue"**.