Python

Data Types & Operators

Output

`print()` : built-in function in python that display input values as text in the output and goes to a new line

Data types

- Integers
- Floats
- Booleans
- Strings

Arithmetic Operator

- `+` : Addition
- `-` : subtraction
- `*` : Multiplication
- `/` : Division
- `%` : mod (the reminder after dividing
- `**` : Exponentiation (note that `^` does not do this operation, as you might have seen in other languages)
- `// :` Divides and rounds down to the nearest integer

Bitwise Operators

- **x << y** : Returns x with the bits shifted to the left by y places (and new bits on the right-hand-side are zeros). This is the same as multiplying x by 2**y.
- **x >> y** : Returns x with the bits shifted to the right by y places. This is the same as //'ing x by 2**y.
- **x & y** : Does a "bitwise and". Each bit of the output is 1 if the corresponding bit of x AND of y is 1, otherwise it's 0.
- **x | y** : Does a "bitwise or". Each bit of the output is 0 if the corresponding bit of x AND of y is 0, otherwise it's 1.
- **~ x** : Returns the complement of x - the number you get by switching each 1 for a 0 and each 0 for a 1. This is the same as -x - 1.
- **x ^ y** : Does a "bitwise exclusive or". Each bit of the output is the same as the corresponding bit in x if that bit in y is 0, and it's the complement of the bit in x if that bit in y is 1.

Variables

Variables names should be **descriptive** of the values they hold so:

- Only use ordinary letters, numbers, and underscores in your variable names. They can't have spaces and need to start with a letter or underscore.
- You can't use Python's **reserved words,** or **"keywords,"** as variable names.
- The pythonic way to name variables is to use all **lowercase** letters and **underscores** to separate words

Assignment Operators

=, += , -=, *= , /=, %=, **=

Comparison Operators

==, !=, >, <, >=, <=

Logical Operators

and, or, not

Special Operators

- Identity Operators
- Membership Operators

Identity Operators

In Python, `is` and `is not` are used to check if two values are located on the same part of the memory. Two variables that are equal do not imply that they are identical.

- **is** : True if only operands are identical (*refers to the same object*)
- **is not** : if operands are not identical

Membership Operators

In Python, `in` and `not in` are the membership operators. They are used to test whether a value or variable is found in a sequence (string, list, tuple, set, and dictionary).

In a dictionary, we can only test for the presence of a key, not the value.

- **in** : True if value is found in sequence
- **not in** : True if value is not found in sequence

Comments

There are 2 types of comments in python:

- One line comment using `# comment`
- Multi line comment using `"""comment """`

Strings

Strings in python are surrounded by either single quotation marks, or double quotation marks
Multiline string we can assign multiline string with using of **3 double or single quotation**

```
a = """Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua."""
print(a)
## output
Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua.
```

The string is an **array of bytes** that stores Unicode characters so this we can access any char in string using it's **index**

```
a = "Hello, World!"
print(a[1])
##output
e
```

Looping through strings

```
for x in "banana":
  print(x)
## output
b
a
n
a
n
a
```

Length of string

So we can use `len()` function

```
a = "Hello, World!"
print(len(a))

##ouput
13
```

Check string

If we need to check if the phrase in string or not we can use `in` & `not in`

Slicing

We can return range of characters by using the slice syntax

```
b = "Hello, World!"
print(b[2:5])
print(b[:5])
print(b[2:])

##ouput
Llo
Hello
llo, World!
```

Concatenate & Repeating

- **+** : for concatenation 2 strings
- ***** : for repeating string number of times

```
s = 'hello'
t = ' world'

print(s + t)
print(s*3)

##ouput
hello world
hellohellohello
```

Modify string

Upper Case `upper()`

```
a = "Hello, World!"
print(a.upper())

## output
```

```
HELLO, WORLD!
```

Lower Case `lower()`

```
a = "Hello, World!"
print(a.lower())

##output
hello, world!
```

Remove whitespace `strip()`

```
a = " Hello, World! "
print(a.strip()) # returns "Hello, World!"
```

Replace string `replace()`

```
a = "Hello, World!"
print(a.replace("H", "J")) # Jello, World!
```

Split string `split()`

This method is used to split a string into substrings by passing a **delimiter & max splits** and returning a list of substrings

```
a = "Hello, World!"
print(a.split(",")) # returns ['Hello', ' World!']
```

Format `format()`

`This method takes the passed arguments, formats them, and places them in the string where placeholders {} are`

```
age = 36
txt = "My name is John, and I am {}"
print(txt.format(age))   # My name is John, and I am 36
```

Join `join()`

Joins the elements of an iterable to the end of the string

```
myTuple = ("John", "Peter", "Vicky")

x = "#".join(myTuple)

print(x)     # John#Peter#Vicky
```

Partition `partition()`
Returns a tuple where the string is parted into three parts

```
txt = "I could eat bananas all day"

x = txt.partition("bananas")

print(x)          # ('I could eat ', 'bananas', ' all day')
```

Collections

- **List** is a collection which is ordered and changeable. Allows duplicate members.
- **Tuple** is a collection which is ordered and unchangeable. Allows duplicate members.
- **Set** is a collection which is unordered, unchangeable*, and unindexed. No duplicate members.
- **Dictionary** is a collection which is ordered** and changeable. No duplicate members.

List

- **Lists** are used to store multiple items in a single variable
- **Lists** are created using square brackets
- Characteristics:
    - Mutable
    - Ordered
    - Duplicates allows
    - Using []

```
thislist = ["apple", "banana", "cherry"]
```

Length of list

Getting length of list using `len(list_name)`

```
thislist = ["apple", "banana", "cherry"]
len(thislist) #
```

Access Items

List items are indexed so we can access it using index

```
thislist = ["apple", "banana", "cherry"]
print(thislist[1]) # banana
print(thislist[-1]) # cherry
print(thislist[1:3]) # ['banana', 'cherry']
```

Change Range of values

To change the value of items within a specific range, define a list with the new values, and refer to the range of index numbers where you want to insert the new values

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "mango"]
thislist[1:3] = ["blackcurrant", "watermelon"]
print(thislist)   # ['apple', 'blackcurrant', 'watermelon', 'orange', 'kiwi', 'mango']
```

If you insert **more** items than you replace, the new items will be **inserted** where you specified, and the remaining items will move accordingly:

```
thislist = ["apple", "banana", "cherry"]
thislist[1:2] = ["blackcurrant", "watermelon"]
print(thislist)  # ['apple', 'blackcurrant', 'watermelon', 'cherry']
```

If you insert **less** items than you replace, the new items will be **inserted** where you specified, and the remaining items will **move** accordingly:

```
thislist = ["apple", "banana", "cherry"]
thislist[1:3] = ["watermelon"]
print(thislist)   # ['apple', 'watermelon']
```

Insert Items

To **insert items** in list we use `insert(index, item)` function

```
thislist = ["apple", "banana", "cherry"]
thislist.insert(2, "watermelon")
print(thislist) # ['apple', 'banana', 'watermelon', 'cherry']
```

To **append item** at the **end** of the list using method `append(item)`

```
fruits = ["apple", "banana", "cherry"]
print("before appending: {}".format(fruits))
fruits.append("orange")
print("after appending: {}".format(fruits))

##ouput
before appending: ['apple', 'banana', 'cherry']
after appending: ['apple', 'banana', 'cherry', 'orange']
```

To **append** elements from another **iterable** object (set, tuple, dict) we use

```
extend(iterable object)
```

```
thislist = ["apple", "banana", "cherry"]
thistuple = ("kiwi", "orange")
thislist.extend(thistuple)
print(thislist)

## output
['apple', 'banana', 'cherry', 'kiwi', 'orange']
```

Remove Elements

`remove()` method removes specific element

```
thislist = ["apple", "banana", "cherry"]
thislist.remove("banana")
print(thislist)         # ['apple', 'cherry']
```

`pop()` method remove specific index

```
thislist = ["apple", "banana", "cherry"]
thislist.pop(1)
print(thislist)         # ['apple', 'cherry']
```

`del()` method delete specific index or completely delete list

```
thislist = ["apple", "banana", "cherry"]
del thislist[0]
print(thislist)         # ['banana', 'cherry']
del thislist            #now is deleted and thislist become undefined
```

Clear list

Using `clear()` method

```
thislist = ["apple", "banana", "cherry"]
thislist.clear()
print(thislist)         # []
```

Loop through lists

```
thislist = ["apple", "banana", "cherry"]
for x in thislist:
  print(x)
```

```
thislist = ["apple", "banana", "cherry"]
for i in range(len(thislist)):
  print(thislist[i])
```

```
thislist = ["apple", "banana", "cherry"]
[print(x) for x in thislist]
```

Another form that we used it more with `input()` to **`split`** `input` and **`cast`** `it` to **`variables`**

```
newlist = [expression for item in iterable if condition == True]
```

min & max

We use `min(list_name)` to get min element in list (only if all elements in list have same type) &
`max(list_name)` to get max element in list

count

`list_name.count(value)` Returns the number of elements with the specified value

Sort List

We can use it's own function `list_name.sort()` in this case the list will be sorted otherwise we can use General function `sorted(list_name)` it will return another list that have elements of our list but sorted

```
thislist = ["orange", "mango", "kiwi", "pineapple", "banana"]
thislist.sort()
print(thislist)              # ['banana', 'kiwi', 'mango', 'orange', 'pineapple']
```

```
thislist = ["orange", "mango", "kiwi", "pineapple", "banana"]
print(sorted(thislist))      # ['banana', 'kiwi', 'mango', 'orange', 'pineapple']
```

If we want to sort descending we use `list_name.sort(reverse = True)` or `sorted(list_name, reverse = True)`

Customize sort function

We can make that by passing keyword `key = function`

```
def myfunc(n):
  return abs(n - 50)

thislist = [100, 50, 65, 82, 23]
thislist.sort(key = myfunc)
print(thislist)
```

```
thislist = ["banana", "Orange", "Kiwi", "cherry"]
thislist.sort(key = str.lower)
print(thislist)
```

Reverse list

To reverse list we use method `list_name.reverse()`

```
thislist = ["banana", "Orange", "Kiwi", "cherry"]
thislist.reverse()
print(thislist)        # ['banana', 'pineapple', 'kiwi', 'mango', 'orange']
```

Copy List

We cannot make copy list simple by typing `list2 = list1` because list2 will only be **reference** to list1 so any change in list1 will be automatically effect in list2
So there are another way to use `copy()` method

```
thislist = ["apple", "banana", "cherry"]
mylist = thislist.copy()
print(mylist)           # ["apple", "banana", "cherry"]
```

Join two lists

We can make it using '+' or using `extend()` method

Tuple

The same as the list is used to store multiple items in a single variable, it's characteristics:

- Immutable
- Ordered
- Can access elements using index
- Allow Duplicates
- Allow multitypes
- Using ()

```
thistuple = ("apple", "banana", "cherry", "apple", "cherry")
print(thistuple)        # ('apple', 'banana', 'cherry', 'apple', 'cherry')
```

💡Note:
When we created tuple with one value we don't make this `thistuple = ("apple")` because `type` of this is `str`, not `tuple` to solve this we write just ',' after element `thistuple = ("apple", )`

Update tuple

Once we create tuple we can't update it because it's not emmutable so if we need to remove or add elements or update we just:
1- convert tuple to list
2- do operation (insert, update, remove)
3- convert back to tuple

💡Note:
Another way to add elements to tuple we create new tuple that content elements we want to insert them we add this tuple to exist tuple

```
thistuple = ("apple", "banana", "cherry")
y = ("orange",)
thistuple += y

print(thistuple)              # ('apple', 'banana', 'cherry', 'orange')
```

Packing & Unpacking

Packing : this process when we assign values to tuple
Unpacking : this process when extract this values from tuble to variables or lists

```
fruits = ("apple", "banana", "cherry")          #packing

green, yellow, red = fruits         #unpacking

print(green)            # apple
print(yellow)           # banana
print(red)        # cherry
```

Using Asterisk *
We use it to unpack element then assign them to the variable as `list`

```
fruits = ("apple", "banana", "cherry", "strawberry", "raspberry")

(green, yellow, *red) = fruits

print(green)            # apple
print(yellow)           # banana
print(red)              # ['cherry', 'strawberry', 'raspberry']
```

💡Note:
We can use '+' operator & '*' operator as it's in strings

Count

This method returns number of times a specified value occures in tuple

Index

This method searches in tuble for specified value and return the position of where it was found

Set

Sets are used to store multiple values in one variable so it's characteristics:

- Immutable
- Unorderd
- Duplicates not allowed
- Using {}

```
len(set_name)
```

This method to get set length

```
add()
```

This method add element to set

```
update(any iteratable)
```

This method is use to add elements of any iteratble to set

```
remove(ele)
```

This method to remove specified element from set

```
pop()
```

This method to remove last element in set but set is unordered so we don't know what is last element

```
clear()
```

This method to clear set

```
del + set_name
```

To delete set

```
union(another_list)
```

Use to join to sets to gothers similar to **update** function

```
intersection_update(another_list)
```

This method will keep only items that are present in both sets

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}

x.intersection_update(y)
print(x)                # {'apple'}
```

`intersection()`

method will return a new set, that only contains the items that are present in both sets.

`symmetric_difference_update()`

method will keep only the elements that are NOT present in both sets.

`symmetric_difference()`

method will return a new set, that contains only the elements that are NOT present in both sets.

Dictionary

Dictionaries are used to store data values in key: value pairs
Characteristics:

- Mutable
- Ordered
- Duplicates not allowed
- Using {}

Accessing

There are 2 ways:
1- using [] like this `dic[key]`
2- using `get(key) and we can also pass value that will be returned if key not found`

Keys & values

keys() method return list of keys
values() method return list of values

Adding

To add elements we can use update function or we make `dic[key] = value`

Remove

To remove an element from dict we use pop() method

💡Note:
Dictionary can contains another dictionary we name that **nested dictionary**

Control flow

Conditions

If, elif, else

```
a = 200
b = 33
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
else:
  print("a is greater than b")
```

Short hand if

```
if a > b: print("a is greater than b")
```

Short Hand If ... Else

```
a = 2
b = 330
print("A") if a > b else print("B")
```

```
a = 330
b = 330
print("A") if a > b else print("=") if a == b else print("B")
```

Loops

While

```
i = 1
while i < 6:
  print(i)
  i += 1
else:
  print("i is no longer less than 6")
```

for

We can use range() function with it starts from 0 be default it accept at least 1 parameter up to 3 parameter

- 1 parameter start from 0 to this parameter (parameter not included)
- 2 parameter start from p1 to p2 (p2 not included)
- 3 parameter start from p1 to p2 (p2 not include) increment by p3

```
for x in range(2, 30, 3):
  print(x)
```

Zip & Enumerate

`zip` and `enumerate` are useful built-in functions that can come in handy when dealing with loops.

Zip

returns an iterator that combines multiple iterables into one sequence of tuples. Each tuple contains the elements in that position from all the iterables

```
letters = ['a', 'b', 'c']
nums = [1, 2, 3]

for letter, num in zip(letters, nums):
    print("{}: {}".format(letter, num))
```

For unzipping we use '*'

```
some_list = [('a', 1), ('b', 2), ('c', 3)]
letters, nums = zip(*some_list)    # letters , nums will be tuples
```

Enumerate

is a built in function that returns an iterator of tuples containing indices and values of a list. You'll often use this when you want the index along with each element of an iterable in a loop.

```
letters = ['a', 'b', 'c', 'd', 'e']
for i, letter in enumerate(letters):
    print(i, letter)
```

List comprehensions

```
capitalized_cities = [city.title() for city in cities]
```

```
squares = [x**2 for x in range(9) if x % 2 == 0]
```

```
squares = [x**2 if x % 2 == 0 else x + 3 for x in range(9)]
```

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")
```

Arbitrary Arguments

We use it if we don't know number of parameters that will pass to the function so we use '*' or '**'

'*' : This way the function will receive a tuple of arguments, and can access the items accordingly

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

my_function("Emil", "Tobias", "Linus")
```

'**' : This way the function will receive a dictionary of arguments, and can access the items accordingly

```
def my_function(**kid):
  print("His last name is " + kid["lname"])

my_function(fname = "Tobias", lname = "Refsnes")
```

Lamda

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
```

```
x = lambda a : a + 10
print(x(5))
```

Handling Errors

The `try` block lets you test a block of code for errors.
The `except` block lets you handle the error.
The `else` block lets you execute code when there is no error.
The `finally` block lets you execute code, regardless of the result of the try- and except blocks.

```
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")
```

Raise an exception

As a Python developer you can choose to throw an exception if a condition occurs.
To throw (or raise) an exception, use the `raise` keyword.

```
x = -1

if x < 0:
  raise Exception("Sorry, no numbers below zero")
```

Files

File Handling

File handling is an important part of any web application.
Python has several functions for creating, reading, updating, and deleting files.

The **`open()`** function takes two parameters; filename, and mode.

"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

"t" - Text - Default value. Text mode
"b" - Binary - Binary mode (e.g. images)

```
f = open("demofile.txt")
```

```
f = open("demofile.txt", "rt")
```

Reading File

- **`read()`**
- **`readline()`**
- looping on handle (object)

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

readline()
return one line

looping on handle
We can get lines on file one by one with looping on it using handle (object)

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

Write & Creating Files

Write to an Existing File

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

Create a New File

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 exist
"a" - Append - will create a file if the specified file does not exist
"w" - Write - will create a file if the specified file does not exist

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")
```

With

allows you to open a file, do operations on it, and automatically close it after the indented code is executed, in this case, reading from the file. Now, we don't have to call f.close()! You can only access the file object, f, within this indented block.

```
with open('my_path/my_file.txt', 'r') as f:
    file_data = f.read()
```

Scripting

We can actually `import` Python code from other scripts, which is helpful if you are working on a bigger project where you want to organize your code into multiple files and reuse code in those files

It's the standard convention for `import` statements to be written at the top of a Python script, each one on a separate line. This `import` statement creates a module object called `useful_functions`. Modules are just Python files that contain definitions and statements. To access objects from an imported module, you need to use dot notation.

```
import useful_functions
```

We can **alias** module name with using `as` keyword

```
import useful_functions as uf
uf.add_five([1, 2, 3, 4])
```

Using main block

To avoid running executable statements in a script when it's imported as a module in another script, include these lines in an `if __name__ == "__main__"` block. Or alternatively, include them in a function called main() and call this in the if main block.

Whenever we run a script like this, Python actually sets a special built-in variable called `__name__` for any module. When we run a script, Python recognizes this module as the main program, and sets the`__name__`variable for this module to the string `"__main__"`. For any modules that are imported in this script, this built-in `__name__` variable is just set to the name of that module. Therefore, the condition `if __name__ == "__main__"`is just checking whether this module is the main program.

```
# demo.py

import useful_functions as uf

scores = [88, 92, 79, 93, 85]

mean = uf.mean(scores)
curved = uf.add_five(scores)

mean_c = uf.mean(curved)

print("Scores:", scores)
print("Original Mean:", mean, " New Mean:", mean_c)

print(__name__)
print(uf.__name__)
```

```
# useful_functions.py

def mean(num_list):
    return sum(num_list) / len(num_list)

def add_five(num_list):
    return [n + 5 for n in num_list]

def main():
    print("Testing mean function")
    n_list = [34, 44, 23, 46, 12, 24]
    correct_mean = 30.5
    assert(mean(n_list) == correct_mean)

    print("Testing add_five function")
    correct_list = [39, 49, 28, 51, 17, 29]
    assert(add_five(n_list) == correct_list)

    print("All tests passed!")

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

Standard Modules

Techniques for Importing Modules

1-To import an individual function or class from a module

```
from module_name import object_name
```

2-To import multiple individual objects from a module

```
from module_name import first_object, second_object
```

3- To rename a module

```
import module_name as new_name
```

4-To import an object from a module and rename it

```
from module_name import object_name as new_name
```

5-To import every object individually from a module (DO NOT DO THIS)

```
from module_name import *
```

6- If you really want to use all of the objects from a module, use the standard import module_name statement instead and access each of the objects with the dot notation.

```
import module_name
```

Modules, packages and Names

In order to manage the code better, modules in the Python Standard Library are split down into sub-modules that are contained within a package. A **package** is simply a module that contains sub-modules. A sub-module is specified with the usual dot notation.

Modules that are submodules are specified by the package name and then the submodule name separated by a dot. You can import the submodule like this.

```
import package_name.submodule_name
```

Iterators and Generators

Iterables are objects that can return one of their elements at a time, such as a list. Many of the built-in functions we've used so far, like 'enumerate,' return an iterator.

An **iterator** is an object that represents a stream of data. This is different from a **list**, which is also an iterable, but is not an iterator because it is not a stream of data.

Generators are a simple way to create iterators using functions. You can also define iterators using **classes**, which you can read more about here.

Here is an example of a generator function called `my_range`, which produces an iterator that is a stream of numbers from 0 to (x - 1).

```
def my_range(x):
    i = 0
    while i < x:
        yield i
        i += 1
```

Notice that instead of using the return keyword, it uses `yield`. This allows the function to return values one at a time, and start where it left off each time it's called. This `yield` keyword is what differentiates a generator from a typical function.

OOP

Classes & Objects

Python is an object-oriented programming language.
Almost everything in Python is an object, with its properties and methods.
A Class is like an object constructor or a "blueprint" for creating objects.

Create class

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

Create instance (object)

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

__init__()

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

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

p1 = Person("John", 36)

print(p1.name)
print(p1.age)
```

💡Note:

- Any **attribute** in **class** level belongs to **class** `'static'`
- We use `self` with any method related to the **instance** to point to an **instance**
- If we want to create **class** belong to **class** we just use **decorator** `@classmethod` before method and we just make first arg in this method is `cls` which point to class
- If we want to make static method we use **decorator** `@staticmethod` before method and in this case, we don't need to pass any argument pointing to **class** or **instance**
- We use **keyword** `pass` if we need to make an empty class
- Using **method** `dir()` if we need to know all **attributes** and **methods** that this **class** have
- **Encapsulation** means bundling data with methods that operate on data

__str()__

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

Implementation: str

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

  def __str__(self):
    cust_str = """
    Customer:
      name: {name}
      balance: {balance}
    """.format(name = self.name, \
               balance = self.balance)
    return cust_str
```

```
cust = Customer("Maryam Azar", 3000)

# Will implicitly call __str__()
print(cust)
```

```
Customer:
  name: Maryam Azar
  balance: 3000
```

__reper()__

compute the “official” string representation of an object (a representation that has all information about the object) and str() is used to compute the “informal” string representation of an object (a representation that is useful for printing the object).

Implementation: repr

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

  def __repr__(self):
    # Notice the '...' around name
    return "Customer('{name}', {balance})".format(name = self.name, balance = self.balance)
cust = Customer("Maryam Azar", 3000)
cust # <--- # Will implicitly call __repr__()
```

```
Customer('Maryam Azar', 3000) # <--- not Customer(Maryam Azar, 3000)
```

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 a base class.
Child class is a class that inherits from another class, also called a derived class.

```
class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)
```

Make child inherit person

```
class Student(Person):
  pass
```

Usign Constructor

```
class Student(Person):
  def __init__(self, fname, lname):
    Person.__init__(self, fname, lname)
```

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

Add new properities after inheritance

```
class Student(Person):
  def __init__(self, fname, lname):
    super().__init__(fname, lname)
    self.graduationyear = 2019
```

Operator overloading

We can use operator overloading with using of this magic methods

Operator	Method
`==`	`__eq__()`
`!=`	`__ne__()`
`>=`	`__ge__()`
`<=`	`__le__()`
`>`	`__gt__()`
`<`	`__lt__()`

Exceptions

Exceptions Type

```
BaseException
+-- Exception
    +-- ArithmeticError                    # <---
    |    +-- FloatingPointError
    |    +-- OverflowError
    |    +-- ZeroDivisionError             # <---
    +-- TypeError
    +-- ValueError
    |    +-- UnicodeError
    |         +-- UnicodeDecodeError
    |         +-- UnicodeEncodeError
    |         +-- UnicodeTranslateError
    +-- RuntimeError
   ...
+-- SystemExit
...
```

```
class BalanceError(Exception): pass
```

```
class Customer:
   def __init__(self, name, balance):
    if balance < 0 :
       raise BalanceError("Balance has to be non-negative!")
    else:
       self.name, self.balance = name, balance
```

```
cust = Customer("Larry Torres", -100)
```

```
Traceback (most recent call last):
  File "script.py", line 11, in <module>
    cust = Customer("Larry Torres", -100)
  File "script.py", line 6, in __init__
    raise BalanceError("Balance has to be non-negative!")
BalanceError: Balance has to be non-negative!
```