PYTHON BASICS

Python with exampls

Abstract

This document explains the basic concepts of core python with examples

Haramohan Sahu

hara.sahu@gmail.com

Contents

Python Architecture

Let's now talk about Python architecture and its usual flow –

i. Parser

It uses the source code to generate an abstract syntax tree.

ii. Compiler

It turns the abstract syntax tree into Python bytecode.

iii. Interpreter

It executes the code line by line in a REPL (Read-Evaluate-Print-Loop) fashion. On Windows, when you want to run the Python interpreter in the shell, you can type the following:
$python

Python Frameworks

i. Django

python Django - Open source python projects

Python Django is a free and open-source framework written in Python and is the most common framework for Python. It allows you to create database-driven websites. It follows the DRY Principle (Don't Repeat Yourself). This is a design philosophy that keeps code simple and eloquent.

Popular websites like Instagram, Mozilla, and Disqus make use of it.

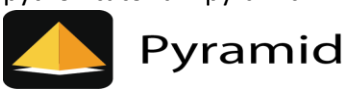
ii. Flask

flask logo

Like Django, Flask is a web framework written in Python itself. It is a micro framework because it does not need certain libraries and tools. It also does not have form validation or a database abstraction layer. However, you can make use of extensions to add extra features.

iii. Pyramid

python tutorial - pyramid



Pyramid is another web framework. It is neither a mega-framework that would make decisions for you nor a micro-framework that wouldn't force decisions. It gives you optimal liberty of your project.

iv. Tornado

python tornado

Another open-source web framework, Tornado is written in Python Language. It is noted for its excellent performance and scalability.

v. Bottle

Python bottle logo

Like Flask, it is a micro-framework for Python. It is used for web development. Bottle is known for its speed, simplicity, and lightweight. A single file can run both Python 2.5+ and 3.x.

vi. web2py

python programming - web2py logo

Written in Python, web2py is another open source web framework. It emphasizes on rapid development and follows an MVC architecture. MVC stands for Model View Controller.

vii. NumPy

Python Machine Learning Environment Setup

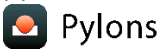
NumPy is an open-source framework for Python. We use it for scientific computing. It supports large multidimensional arrays and matrices, and functions to operate on them.

viii. SciPy

SciPy is a Python library that you can use for scientific computing. It has modules for linear algebra, interpolation, fast Fourier transform(FFT), image processing, and many more. It uses multidimensional arrays from the NumPy module.

ix. Pylons

python tutorial - pylons logo



This is a deprecated framework, which means it is no longer recommended. It is a web framework and is open source as well. It makes extensive use of third-party tools.

Big Companies Using Python

Many big names use (or have used) Python for their products/services. Some of these are:

- NASA
- Google
- Nokia
- IBM
- Yahoo! Maps
- Walt Disney Feature Animation
- Facebook
- Netflix
- Expedia
- Reddit
- Quora
- MIT
- Disqus
- Hike
- Spotify
- Udemy
- Shutterstock
- Uber
- Amazon
- Mozilla
- Dropbox
- Pinterest
- Youtube

Python Datatypes:

1. Number
2. List
3. Tuple
4. String
5. Set
6. Dictionary

Number Data Type in Python:

Python supports four different numerical types

1. int (signed integers) -> positive or negative whole numbers with no decimal point.
2. long (long integers)-> are integers of unlimited size, written like integers and followed by an uppercase or lowercase L
3. float (floating point real values)-> written with a decimal point dividing the integer and fractional parts.
4. complex (complex numbers)

They are defined as int, float, and complex classes in Python. Integers and floating points are separated by the presence or absence of a decimal point.

For instance,
5 is an integer whereas 5.0 is a floating-point number.
Complex numbers are written in the form, x + yj, where x is the real part and y is the imaginary part.

We can use the **type**() function to know which class a variable or a value belongs to and **isinstance**() function to check if it belongs to a particular class.
Let's look at an example:

```
a = 5
b=5.5
print(type(a))
print(type(5.0))
c = 5 + 3j
print(type(c))
print(c)
print(isinstance(c, complex))
print(isinstance(a, int))
print(isinstance(b, float))
```

Output:

```
<class 'int'>
<class 'float'>
<class 'complex'>
(5+3j)
True
True
True
```

While integers can be of any length, a floating-point number is accurate only up to 15 decimal places (the 16th place is inaccurate).
The numbers we deal with every day are of the decimal (base 10) number system.

Number System	**Prefix**
Binary	'0b' or '0B'
Octal	'0o' or '0O'
Hexadecimal	'0x' or '0X'

Here are some examples

```
# Output: 107
print(0b1101011)

# Output: 253 (251 + 2)
print(0xFB + 0b10)

# Output: 13
print(0o15)
```

Type Conversion

We can convert one type of number into another. This is also known as coercion.

Operations like addition, subtraction coerce integer to float implicitly (automatically), if one of the operands is float.

1 + 2.0
Output:

3.0

We can see above that 1 (integer) is coerced into 1.0 (float) for addition and the result is also a floating point number.
We can also use built-in functions like int(), float() and complex() to convert between types explicitly. These functions can even convert from strings.

int(2.3)
2
int(-2.8)
-2
float(5)
5.0
complex('3+5j')
(3+5j)
When converting from float to integer, the number gets truncated (decimal parts are removed).

Some in built function:

pow(x, y)
The value of x**y.
round(x [,n])
x rounded to n digits from the decimal point.

Python Decimal

We all know that the sum of 1.1 and 2.2 is 3.3, but Python seems to disagree.
print((1.1 + 2.2) == 3.3)
Output:
False
What is going on?

It turns out that floating-point numbers are implemented in computer hardware as binary fractions as the computer only understands binary (0 and 1).
Due to this reason, most of the decimal fractions we know, cannot be accurately stored in our computer.
Let's take an example. We cannot represent the fraction 1/3 as a decimal number. This will give 0.33333333...
which is infinitely long, and we can only approximate it, but never be equal. Hence, it is the limitation of our computer hardware and not an error in Python.
To overcome this issue, we can use the decimal module that comes with Python. While floating-point numbers have precision up to 15 decimal places,
the decimal module has user-settable precision.

Let's see the difference:
import decimal
print(0.1)

```
print(decimal.Decimal(0.1))
```
Output:
```
0.1
0.1000000000000000055511151231257827021181583404541015625
```

Some more example:

```
from decimal import Decimal as D
print(D('1.1') + D('2.2'))
print(D('1.2') * D('2.50'))
```

output:

```
3.3
3.000
```

Notice the trailing zeroes in the above example.

We might ask, why not implement Decimal every time, instead of float? The main reason is efficiency. Floating point operations are carried out must faster than Decimal operations.

Python Fractions:

Python provides operations involving fractional numbers through its fractions module.
A fraction has a numerator and a denominator, both of which are integers. This module has support for rational number arithmetic.
We can create Fraction objects in various ways. Let's have a look at them.

```
import fractions

print(fractions.Fraction(1.5))
print(fractions.Fraction(5))
print(fractions.Fraction(1,3))
```

Output

```
3/2
5
1/3
```

While creating Fraction from float, we might get some unusual results. This is due to the imperfect binary floating point number representation as discussed in the previous section.

Fortunately, Fraction allows us to instantiate with string as well. This is the preferred option when using decimal numbers.

```
import fractions
from fractions import Fraction as F
```

```
# As float
# Output: 2476979795053773/2251799813685248
print(fractions.Fraction(1.1))

# As string
# Output: 11/10
print(fractions.Fraction('1.1'))


# Output: 11/10
print(fractions.Fraction('1.1'))
print(F(1, 3) + F(1, 3))
print(1 / F(5, 6))
print(F(-3, 10) > 0)
print(F(-3, 10) < 0)
```

Output

```
2476979795053773/2251799813685248
11/10
2/3
6/5
False
True
```

Python List

List is one of the most frequently used and very versatile data types used in Python.
items in a list need not be of the same type.

How to create a list?

In Python programming, a list is created by placing all the items (elements) inside square brackets [], separated by commas.

It can have any number of items and they may be of different types (integer, float, string etc.).

```
# empty list
my_list = []

# list of integers
my_list = [1, 2, 3]

# list with mixed data types
my_list = [1, "Hello", 3.4]
```

A list can also have another list as an item. This is called a nested list.

```
# nested list
my_list = ["mouse", [8, 4, 6], ['a']]
```

How to access elements from a list?

List Index

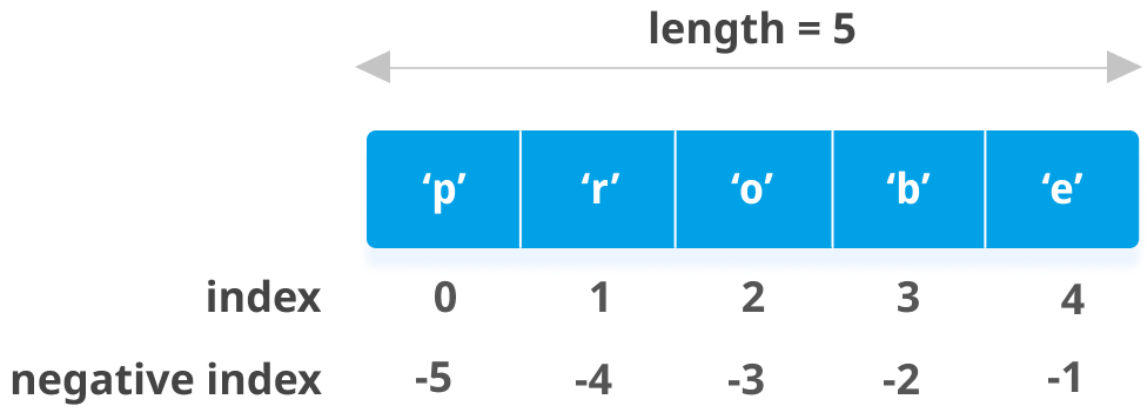
We can use the index operator [] to access an item in a list. In Python, indices start at 0.
So, a list having 5 elements will have an index from 0 to 4.
Trying to access indexes other than these will raise an IndexError. The index must be an integer. We can't use float or other types, this will result in TypeError.

```
# List indexing

my_list = ['p', 'r', 'o', 'b', 'e']

# Output: p
print(my_list[0])

# Output: o
print(my_list[2])

# Output: e
print(my_list[4])

# Nested List
n_list = ["Happy", [2, 0, 1, 5]]

# Nested indexing
print(n_list[0][1])

print(n_list[1][3])

# Error! Only integer can be used for indexing
print(my_list[4.0])
```

Negative indexing

Python allows negative indexing for its sequences. The index of -1 refers to the last item, -2 to the second last item and so on.

```
# Negative indexing in lists
my_list = ['p','r','o','b','e']
print(my_list[-1])
print(my_list[-5])
```



Accessing Values in Lists

To access values in lists, use the square brackets for slicing along with the index or indices to obtain value available at that index. For example –

```
#!/usr/bin/python

list1 = ['physics', 'chemistry', 1997, 2000];
list2 = [1, 2, 3, 4, 5, 6, 7 ];
print "list1[0]: ", list1[0]
print "list2[1:5]: ", list2[1:5]
```

Output

```
list1[0]:  physics
list2[1:5]:  [2, 3, 4, 5]
```

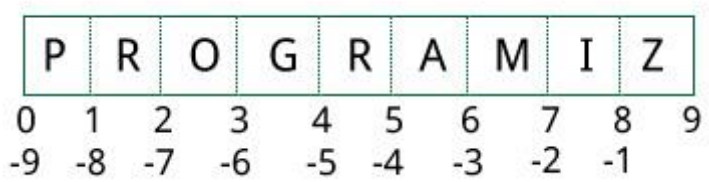
How to slice lists in Python?
We can access a range of items in a list by using the slicing operator :(colon).

```
# List slicing in Python
my_list = ['p','r','o','g','r','a','m','i','z']
# elements 3rd to 5th
print(my_list[2:5])
# elements beginning to 4th
print(my_list[:-5])
# elements 6th to end
print(my_list[5:])
# elements beginning to end
print(my_list[:])
```

output:

```
['o', 'g', 'r']
['p', 'r', 'o', 'g']
['a', 'm', 'i', 'z']
['p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z']
```

Slicing can be best visualized by considering the index to be between the elements as shown below. So if we want to access a range, we need two indices that will slice that portion from the list.



How to change or add elements to a list?

Lists are mutable, meaning their elements can be changed unlike string or tuple.
We can use the assignment operator (=) to change an item or a range of items.

```
# Correcting mistake values in a list
odd = [2, 4, 6, 8]
# change the 1st item
odd[0] = 1
print(odd)
# change 2nd to 4th items
odd[1:4] = [3, 5, 7]
print(odd)
```

Output

```
[1, 4, 6, 8]
[1, 3, 5, 7]
```

We can add one item to a list using the append() method or add several items using extend() method.

```
# Appending and Extending lists in Python
odd = [1, 3, 5]

odd.append(7)

print(odd)

odd.extend([9, 11, 13])

print(odd)
```

Output

```
[1, 3, 5, 7]
[1, 3, 5, 7, 9, 11, 13]
```

We can also use + operator to combine two lists. This is also called concatenation.
The * operator repeats a list for the given number of times.

```
# Concatenating and repeating lists
odd = [1, 3, 5]
print(odd + [9, 7, 5])
```

```
print(["re"] * 3)
```

Output

```
[1, 3, 5, 9, 7, 5]
['re', 're', 're']
```

Furthermore, we can insert one item at a desired location by using the method insert() or insert multiple items by squeezing it into an empty slice of a list.

```
# Demonstration of list insert() method
odd = [1, 9]
odd.insert(1,3)

print(odd)
odd[2:2] = [5, 7]
print(odd)
```

output:

```
[1, 3, 9]
[1, 3, 5, 7, 9]
```

Python List Methods

append() - Add an element to the end of the list
extend() - Add all elements of a list to the another list
insert() - Insert an item at the defined index
remove() - Removes an item from the list
pop() - Removes and returns an element at the given index
clear() - Removes all items from the list
index() - Returns the index of the first matched item
count() - Returns the count of the number of items passed as an argument
sort() - Sort items in a list in ascending order
reverse() - Reverse the order of items in the list
copy() - Returns a shallow copy of the list

How to delete or remove elements from a list?

We can delete one or more items from a list using the keyword del. It can even delete the list entirely.

```
# Deleting list items
my_list = ['p', 'r', 'o', 'b', 'l', 'e', 'm']

# delete one item
del my_list[2]

print(my_list)

# delete multiple items
del my_list[1:5]

print(my_list)

# delete entire list
del my_list
```

```
# Error: List not defined
print(my_list)
```

Output

```
['p', 'r', 'b', 'l', 'e', 'm']
['p', 'm']
Traceback (most recent call last):
  File "<string>", line 18, in <module>
NameError: name 'my_list' is not defined
```

We can use remove() method to remove the given item or pop() method to remove an item at the given index.

The pop() method removes and returns the last item if the index is not provided. This helps us implement lists as stacks (first in, last out data structure).

We can also use the clear() method to empty a list.

```
my_list = ['p','r','o','b','l','e','m']
my_list.remove('p')

# Output: ['r', 'o', 'b', 'l', 'e', 'm']
print(my_list)

# Output: 'o'
print(my_list.pop(1))

# Output: ['r', 'b', 'l', 'e', 'm']
print(my_list)

# Output: 'm'
print(my_list.pop())

# Output: ['r', 'b', 'l', 'e']
print(my_list)

my_list.clear()

# Output: []
print(my_list)
```

Finally, we can also delete items in a list by assigning an empty list to a slice of elements.

```
>>> my_list = ['p','r','o','b','l','e','m']
my_list[2:3] = []
Print( my_list)
[]
my_list = ['p','r','o','b','l','e','m']
my_list[2:5] = []
print( my_list)
['p', 'r', 'e', 'm']
```

Some examples of Python list methods:

```
# Python list methods
my_list = [3, 8, 1, 6, 0, 8, 4]

# Output: 1 → The first occurrence of the number in the list
print(my_list.index(8))
# Output: 2
print(my_list.count(8))
my_list.sort()

# Output: [0, 1, 3, 4, 6, 8, 8]
print(my_list)
my_list.reverse()
# Output: [8, 8, 6, 4, 3, 1, 0]
print(my_list)
```

Elegant way to create new List

One of the language's most distinctive features is the **list comprehension**, which you can use to create powerful functionality within a single line of code.

There are a few different ways you can create lists in Python.

1. **Using for Loops**

```
>>> squares = []
>>> for i in range(10):
...     squares.append(i * i)
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

2. **Using map() Objects**

map() provides an alternative approach that's based in functional programming. You pass in a function and an iterable, and map() will create an object.
This object contains the output you would get from running each iterable element through the supplied function.

```
>>> txns = [1.09, 23.56, 57.84, 4.56, 6.78]
>>> TAX_RATE = .08
>>> def get_price_with_tax(txn):
...     return txn * (1 + TAX_RATE)
```

```
>>> final_prices = map(get_price_with_tax, txns)
>>> list(final_prices)
[1.1772000000000002, 25.4448, 62.467200000000005, 4.9248, 7.322400000000001]
```

3. Using List Comprehensions

```
>>> squares = [i * i for i in range(10)]
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

This code is equivalent to:

```
pow2 = []
for x in range(10):
   pow2.append(2 ** x)
```

Here just compare the list using the for loop one. we can do it in one line

```
>>> TAX_RATE = .08
>>> def get_price_with_tax(txn):
...     return txn * (1 + TAX_RATE)
>>> final_prices = [get_price_with_tax(i) for i in txns]
>>> final_prices
[1.1772000000000002, 25.4448, 62.467200000000005, 4.9248, 7.322400000000001]
```

The only distinction between this implementation and map() is that the list comprehension in Python returns a list, not a map object

List Compression using conditional Logic:

```
>>> sentence = 'the rocket came back from mars'
>>> vowels = [i for i in sentence if i in 'aeiou']
>>> vowels
['e', 'o', 'e', 'a', 'e', 'a', 'o', 'a']
```

some example:

```
>>> for x in range(10):
...     squares.append(x**2)
...
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

or we can write in one line:

```
squares = list(map(lambda x: x**2, range(10)))
```

or, equivalently:

```
squares = [x**2 for x in range(10)]
```

Like this you can use Set and Dictionary Comprehensions

```
>>> unique_vowels = {i for i in quote if i in 'aeiou'}
>>> unique_vowels
{'a', 'e', 'u', 'i'}
```

In set comprehension, you will not get duplicate values.

Other List Operations in Python

List Membership Test

We can test if an item exists in a list or not, using the keyword in.

```
my_list = ['p', 'r', 'o', 'b', 'l', 'e', 'm']

# Output: True
print('p' in my_list)

# Output: False
print('a' in my_list)

# Output: True
print('c' not in my_list)
```

Output

```
True
False
True
```

Iterating Through a List

Using a for loop we can iterate through each item in a list.

```
for i in ['apple','banana','mango']:
    print("I like",i)
```

Using Lists as Stacks?

The list methods make it very easy to use a list as a stack, where the last element added is the first element retrieved (“LIFO”).
To add an item to the top of the stack, use append(). To retrieve an item from the top of the stack, use pop() without an explicit index.
For example:

```
>>>
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
```

```
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

Using Lists as Queues?

```
from collections import deque
queue = deque(["Eric", "John", "Michael"])
queue.append("Terry")           # Terry arrives
queue.append("Graham")          # Graham arrives
print(queue.popleft() )         # The first to arrive now leaves
print(queue.popleft() )         # The second to arrive now leaves
print(queue  )                  # Remaining queue in order of arrival
print(deque(['Michael', 'Terry', 'Graham']))
```

OUTPUT:

```
Eric
John
deque(['Michael', 'Terry', 'Graham'])
deque(['Michael', 'Terry', 'Graham'])
```

what is map Function?

map() function returns a map object(which is an iterator) of the results after applying the given function to each item of a given iterable (list, tuple etc.)
Returns a list of the results after applying the given function to each item of a given iterable (list, tuple etc.)
Syntax :

```
map(fun, iter)
```

Exampe of Map Function:
1)

```
# Python program to demonstrate working
# of map.

# Return double of n
def addition(n):
    return n + n

# We double all numbers using map()
numbers = (1, 2, 3, 4)
result = map(addition, numbers)
```

```
print(list(result))
```

2) using lambda
We can also use lambda expressions with map to achieve above result.

```
# Double all numbers using map and lambda

numbers = (1, 2, 3, 4)
result = map(lambda x: x + x, numbers)
print(list(result))
```

3)

```
# Add two lists using map and lambda

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

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

Python Tuple

A tuple in Python is similar to a list. The difference between the two is that we cannot change the elements of a tuple once it is assigned whereas we can change the elements of a list.

Creating a Tuple

A tuple is created by placing all the items (elements) inside parentheses (), separated by commas.
A tuple can have any number of items and they may be of different types (integer, float, list, string, etc.).
Example:

```
# Different types of tuples

# Empty tuple
my_tuple = ()
print(my_tuple)

# Tuple having integers
my_tuple = (1, 2, 3)
print(my_tuple)

# tuple with mixed datatypes
my_tuple = (1, "Hello", 3.4)
print(my_tuple)

# nested tuple
my_tuple = ("mouse", [8, 4, 6], (1, 2, 3))
print(my_tuple)
```

what is tuple packing.?

A tuple can also be created without using parentheses. This is known as tuple packing.

```
my_tuple = 3, 4.6, "dog"
print(my_tuple)

# tuple unpacking is also possible
a, b, c = my_tuple

print(a)      # 3
print(b)      # 4.6
print(c)      # dog
```
Output

```
(3, 4.6, 'dog')
3
4.6
dog
```

A tuple with one element:

Creating a tuple with one element is a bit tricky.

Having one element within parentheses is not enough. We will need a trailing comma to indicate that it is, in fact, a tuple.

```
my_tuple = ("hello")
print(type(my_tuple))  # <class 'str'>
print(my_tuple)
# Creating a tuple having one element
my_tuple = ("hello",)
print(type(my_tuple))  # <class 'tuple'>

# Parentheses is optional
my_tuple = "hello",
print(type(my_tuple))  # <class 'tuple'>
```

Access Tuple Elements

1. **Indexing**

We can use the index operator [] to access an item in a tuple, where the index starts from 0.

```
# Accessing tuple elements using indexing
my_tuple = ('p','e','r','m','i','t')

print(my_tuple[0])   # 'p'
print(my_tuple[5])   # 't'

# IndexError: list index out of range
# print(my_tuple[6])

# Index must be an integer
# TypeError: list indices must be integers, not float
```

```
# my_tuple[2.0]

# nested tuple
n_tuple = ("mouse", [8, 4, 6], (1, 2, 3))

# nested index
print(n_tuple[0][3])      # 's'
print(n_tuple[1][1])      # 4
```

Output

```
p
t
s
4
```

2. **Negative Indexing**
Python allows negative indexing for its sequences.

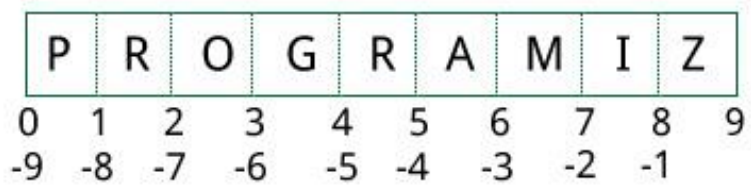
The index of -1 refers to the last item, -2 to the second last item and so on.

```
# Negative indexing for accessing tuple elements
my_tuple = ('p', 'e', 'r', 'm', 'i', 't')

# Output: 't'
print(my_tuple[-1])

# Output: 'p'
print(my_tuple[-6])
```

3. **Slicing**
We can access a range of items in a tuple by using the slicing operator colon :

```
# Accessing tuple elements using slicing
my_tuple = ('p','r','o','g','r','a','m','i','z')

# elements 2nd to 4th
# Output: ('r', 'o', 'g')
print(my_tuple[1:4])

# elements beginning to 2nd
# Output: ('p', 'r') --> from starts to 3 ( as -7 is nothing but 3)
print(my_tuple[:-7])

# elements 8th to end
# Output: ('i', 'z')
print(my_tuple[7:])

# elements beginning to end
# Output: ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')
print(my_tuple[:])
```

Slicing can be best visualized by considering the index to be between the elements as shown below. So if we want to access a range, we need the index that will slice the portion from the tuple.



Changing a Tuple

Unlike lists, tuples are immutable.

This means that elements of a tuple cannot be changed once they have been assigned. But, if the element is itself a mutable data type like list,
its nested items can be changed.

We can also assign a tuple to different values (reassignment).

```
# Changing tuple values
my_tuple = (4, 2, 3, [6, 5])


# TypeError: 'tuple' object does not support item assignment
# my_tuple[1] = 9

# However, item of mutable element can be changed as list is mutable
my_tuple[3][0] = 9    # Output: (4, 2, 3, [9, 5])
print(my_tuple)

# Tuples can be reassigned
my_tuple = ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')

# Output: ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')
print(my_tuple)
```

Deleting a Tuple

As discussed above, we cannot change the elements in a tuple. It means that we cannot delete or remove items from a tuple.

Deleting a tuple entirely, however, is possible using the keyword del.

```
# Deleting tuples
my_tuple = ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')

# can't delete items
# TypeError: 'tuple' object doesn't support item deletion
# del my_tuple[3]

# Can delete an entire tuple
del my_tuple
```

```
# NameError: name 'my_tuple' is not defined
print(my_tuple)
```

Output

```
Traceback (most recent call last):
  File "<string>", line 12, in <module>
NameError: name 'my_tuple' is not defined
```

Tuple method:

Only these 2 methods are available

```
my_tuple = ('a', 'p', 'p', 'l', 'e',)

print(my_tuple.count('p'))  # Output: 2
print(my_tuple.index('l'))  # Output: 3
```

Tuple Membership Test

We can test if an item exists in a tuple or not, using the keyword in.

```
# Membership test in tuple
my_tuple = ('a', 'p', 'p', 'l', 'e',)
# In operation
print('a' in my_tuple)
print('b' in my_tuple)
# Not in operation
print('g' not in my_tuple)
```

Output

```
True
False
True
```

Iterating Through a Tuple

We can use a for loop to iterate through each item in a tuple.

```
# Using a for loop to iterate through a tuple
for name in ('John', 'Kate'):
    print("Hello", name)
```

Output

Hello John

Hello Kate

Advantages of Tuple over List

- We generally use tuples for heterogeneous (different) data types and lists for homogeneous (similar) data types.
- Since tuples are immutable, iterating through a tuple is faster than with list. So there is a slight performance boost.
- Tuples that contain immutable elements can be used as a key for a dictionary. With lists, this is not possible.
- If you have data that doesn't change, implementing it as tuple will guarantee that it remains write-protected.

Python Strings

What is String in Python?

A string is a sequence of characters.

In Python, a string is a sequence of Unicode characters. Unicode was introduced to include every character in all languages and bring uniformity in encoding.

How to create a string in Python?

Strings can be created by enclosing characters inside a single quote or double-quotes. Even triple quotes can be used in Python?

```
# defining strings in Python
# all of the following are equivalent
my_string = 'Hello'
print(my_string)
my_string = "Hello"
print(my_string)
my_string = '''Hello'''
print(my_string)
```

```
# triple quotes string can extend multiple lines
my_string = """Hello, welcome to
        the world of Python"""
print(my_string)
```

How to access characters in a string?

We can access individual characters using indexing and a range of characters using slicing. Index starts from 0. Trying to access a character out of index range will raise an IndexError. The index must be an integer. We can't use floats or other types, this will result into TypeError.

Python allows negative indexing for its sequences.

The index of -1 refers to the last item, -2 to the second last item and so on.
We can access a range of items in a string by using the slicing operator :(colon).

```
#Accessing string characters in Python
str = 'programiz'
print('str = ', str)

#first character
print('str[0] = ', str[0])

#last character
print('str[-1] = ', str[-1])

#slicing 2nd to 5th character
print('str[1:5] = ', str[1:5])

#slicing 6th to 2nd last character
print('str[5:-2] = ', str[5:-2])
```

output:

```
str =  programiz
str[0] =  p
str[-1] =  z
str[1:5] =  rogr
str[5:-2] =  am
```

How to change or delete a string?

Strings are immutable. This means that elements of a string cannot be changed once they have been assigned. We can simply reassign different strings to the same name. because reassign is not modifying the original string object.

```
>>> my_string = 'programiz'
>>> my_string[5] = 'a'
```

```
...
TypeError: 'str' object does not support item assignment
>>> my_string = 'Python'
>>> my_string
'Python'

using del;

>>> del my_string[1]
...
TypeError: 'str' object doesn't support item deletion
>>> del my_string
>>> my_string
...
NameError: name 'my_string' is not defined
```

Python String Operations:

Concatenation of Two or More Strings

```
# Python String Operations
str1 = 'Hello'
str2 ='World!'

# using +
print('str1 + str2 = ', str1 + str2)

# using *
print('str1 * 3 =', str1 * 3)
output
str1 + str2 =  HelloWorld!
str1 * 3 = HelloHelloHello
```

Iterating Through a string

```
# Iterating through a string
count = 0
for letter in 'Hello World':
   if(letter == 'l'):
      count += 1
print(count,'letters found')

output
3 letters found
```

String Membership Test

```
>>> 'a' in 'program'
True
>>> 'at' not in 'battle'
False
```

Built-in functions to Work with Python

Some of the commonly used ones are enumerate() and len(). The enumerate() function returns an enumerate object.
It contains the index and value of all the items in the string as pairs. This can be useful for iteration.

```
str = 'cold'

# enumerate()
list_enumerate = list(enumerate(str))
print('list(enumerate(str) = ', list_enumerate)

#character count
print('len(str) = ', len(str))
list(enumerate(str) =  [(0, 'c'), (1, 'o'), (2, 'l'), (3, 'd')]
len(str) =  4
output
list(enumerate(str) =  [(0, 'c'), (1, 'o'), (2, 'l'), (3, 'd')]
len(str) =  4
```

Python String Formatting

Escape Sequence

If we want to print a text like He said, "What's there?", we can neither use single quotes nor double quotes. This will result in a SyntaxError as the text itself contains both single and double quotes.

```
>>> print("He said, "What's there?"")
...
SyntaxError: invalid syntax
>>> print('He said, "What's there?"')
...
SyntaxError: invalid syntax
```

One way to get around this problem is to use triple quotes. Alternatively, we can use escape sequences. or escape character. "\" but Raw String to ignore escape sequence.

```
>>> print("This is \x61 \ngood example")
```

```
This is a
good example
>>> print(r"This is \x61 \ngood example")
This is \x61 \ngood example
```

The format () Method for Formatting Strings

```
# Python string format () method

# default(implicit) order
default_order = "{}, {} and {}".format('John','Bill','Sean')
print('\n--- Default Order ---')
print(default_order)
output
--- Default Order ---
John, Bill and Sean

Old style formatting
>>> x = 12.3456789
>>> print('The value of x is %3.2f' %x)
The value of x is 12.35
>>> print('The value of x is %3.4f' %x)
The value of x is 12.3457
```

Python Sets

A set is an unordered collection of items. Every set element is unique (no duplicates) and must be immutable (cannot be changed). However, a set itself is mutable. We can add or remove items from it.Sets can also be used to perform mathematical set operations like union, intersection, symmetric difference, etc.

A set is created by placing all the items (elements) inside curly braces {}, separated by comma, or by using the built-in set() function.

It can have any number of items and they may be of different types (integer, float, tuple, string etc.). But a set cannot have mutable elements like lists, sets or dictionaries as its elements.

```
# Different types of sets in Python
# set of integers
my_set = {1, 2, 3}
print(my_set)

# set of mixed datatypes
my_set = {1.0, "Hello", (1, 2, 3)}
```

```
print(my_set)
```
Output

```
{1, 2, 3}
{1.0, (1, 2, 3), 'Hello'}
```

How to create an empty set?

Creating an empty set is a bit tricky.
Empty curly braces {} will make an empty dictionary in Python. To make a set without any elements, we use the set() function without any argument.

```
# Distinguish set and dictionary while creating empty set

# initialize a with {}
a = {}
# check data type of a
print(type(a))
# initialize a with set()
a = set()
# check data type of a
print(type(a))
```

```
<class 'dict'>
<class 'set'>
```

Modifying a set in Python

Sets are mutable. However, since they are unordered, indexing has no meaning.
We cannot access or change an element of a set using indexing or slicing. Set data type does not support it. We can add a single element using the add() method, and multiple elements using the update() method. The update() method can take tuples, lists, strings or other sets as its argument.
In all cases, duplicates are avoided.

```
# initialize my_set
my_set = {1, 3}
print(my_set)

# if you uncomment line 9,
# you will get an error
# TypeError: 'set' object does not support indexing

# my_set[0]

# add an element
# Output: {1, 2, 3}
```

```
my_set.add(2)
print(my_set)

# add multiple elements
# Output: {1, 2, 3, 4}
my_set.update([2, 3, 4])
print(my_set)

# add list and set
# Output: {1, 2, 3, 4, 5, 6, 8}
my_set.update([4, 5], {1, 6, 8})
print(my_set)
```

Output

```
{1, 3}
{1, 2, 3}
{1, 2, 3, 4}
{1, 2, 3, 4, 5, 6, 8}
```

Removing elements from a set

A particular item can be removed from a set using the methods discard() and remove().
The only difference between the two is that the discard() function leaves a set unchanged if the element is not present in the set.
On the other hand, the remove() function will raise an error in such a condition (if element is not present in the set).

```
# Difference between discard() and remove()
# initialize my_set
my_set = {1, 3, 4, 5, 6}
print(my_set)

# discard an element
# Output: {1, 3, 5, 6}
my_set.discard(4)
print(my_set)

# remove an element
# Output: {1, 3, 5}
my_set.remove(6)
print(my_set)

# discard an element
```

```
# not present in my_set
# Output: {1, 3, 5}
my_set.discard(2)
print(my_set)

# remove an element
# not present in my_set
# you will get an error.
# Output: KeyError

my_set.remove(2)
```

Output

```
{1, 3, 4, 5, 6}
{1, 3, 5, 6}
{1, 3, 5}
{1, 3, 5}
Traceback (most recent call last):
  File "<string>", line 28, in <module>
KeyError: 2
```

Similarly, we can remove and return an item using the pop() method.

Since set is an unordered data type, there is no way of determining which item will be popped. It is completely arbitrary.

```
# initialize my_set
# Output: set of unique elements
my_set = set("HelloWorld")
print(my_set)

# pop an element
# Output: random element
print(my_set.pop())
```

Python Set Operations

```
# Set union method
# initialize A and B
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}

# use | operator
# Output: {1, 2, 3, 4, 5, 6, 7, 8}
```

```
print(A | B)
```

or we can use union () method like A.union(B)

```
print(A.union(b))
```

like this , all the mat set operation can be done.

Set Membership Test

We can test if an item exists in a set or not, using the in keyword.

```
# in keyword in a set
# initialize my_set
my_set = set("apple")

# check if 'a' is present
# Output: True
print('a' in my_set)

# check if 'p' is present
# Output: False
print('p' not in my_set)
```

Output

```
True
False
```

Iterating Through a Set

We can iterate through each item in a set using a for loop.

```
>>> for letter in set("apple"):
...     print(letter)
```

Python Frozenset

Frozenset is a new class that has the characteristics of a set, but its elements cannot be changed once assigned.

While tuples are immutable lists, frozensets are immutable sets.

```
# Frozensets
# initialize A and B
A = frozenset([1, 2, 3, 4])
B = frozenset([3, 4, 5, 6])
```

Try these examples on Python shell.

```
>>> A.isdisjoint(B)
False
>>> A.difference(B)
frozenset({1, 2})
>>> A | B
frozenset({1, 2, 3, 4, 5, 6})
>>> A.add(3)
...
AttributeError: 'frozenset' object has no attribute 'add'
```

Python Dictionary

Python dictionary is an unordered collection of items. Each item of a dictionary has a key/value pair.

Dictionaries are optimized to retrieve values when the key is known.

Creating Python Dictionary

Creating a dictionary is as simple as placing items inside curly braces {} separated by commas.

An item has a key and a corresponding value that is expressed as a pair (key: value).

While the values can be of any data type and can repeat, keys must be of immutable type (string, number or tuple with immutable elements) and must be unique.

```
# empty dictionary
my_dict = {}

# dictionary with integer keys
my_dict = {1: 'apple', 2: 'ball'}

# dictionary with mixed keys
my_dict = {'name': 'John', 1: [2, 4, 3]}

# using dict()
my_dict = dict({1:'apple', 2:'ball'})

# from sequence having each item as a pair
my_dict = dict([(1,'apple'), (2,'ball')])
```

As you can see from above, we can also create a dictionary using the built-in dict() function

Accessing Elements from Dictionary

a dictionary uses keys. Keys can be used either inside square brackets [] or with the get() method.

If we use the square brackets [], KeyError is raised in case a key is not found in the dictionary.
On the other hand, the get() method returns None if the key is not found.

```
# get vs [] for retrieving elements
my_dict = {'name': 'Jack', 'age': 26}

# Output: Jack
print(my_dict['name'])

# Output: 26
print(my_dict.get('age'))

# Trying to access keys which doesn't exist throws error
# Output None
print(my_dict.get('address'))

# KeyError
print(my_dict['address'])
```

Output

```
Jack
26
None
Traceback (most recent call last):
  File "<string>", line 15, in <module>
    print(my_dict['address'])
KeyError: 'address'
```

Changing and Adding Dictionary elements

Dictionaries are mutable. We can add new items or change the value of existing items using an assignment operator.
Dictionaries are mutable. We can add new items or change the value of existing items using an assignment operator.

If the key is already present, then the existing value gets updated. In case the key is not present, a new (key: value) pair is added to the dictionary.

```
# Changing and adding Dictionary Elements
```

```
my_dict = {'name': 'Jack', 'age': 26}

# update value
my_dict['age'] = 27

#Output: {'age': 27, 'name': 'Jack'}
print(my_dict)

# add item
my_dict['address'] = 'Downtown'

# Output: {'address': 'Downtown', 'age': 27, 'name': 'Jack'}
print(my_dict)
```

Output

```
{'name': 'Jack', 'age': 27}
{'name': 'Jack', 'age': 27, 'address': 'Downtown'}
```

Removing elements from Dictionary

We can remove a particular item in a dictionary by using the pop() method. This method removes an item with the provided key and returns the value.
The popitem() method can be used to remove and return an arbitrary (key, value) item pair from the dictionary. All the items can be removed at once, using the clear() method.

We can also use the del keyword to remove individual items or the entire dictionary itself.

```
# Removing elements from a dictionary

# create a dictionary
squares = {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

# remove a particular item, returns its value
# Output: 16
print(squares.pop(4))

# Output: {1: 1, 2: 4, 3: 9, 5: 25}
print(squares)

# remove an arbitrary item, return (key,value)
# Output: (5, 25)
print(squares.popitem())

# Output: {1: 1, 2: 4, 3: 9}
```

```
print(squares)

# remove all items
squares.clear()
```

Removing elements from a dictionary

```
# create a dictionary
squares = {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

# remove a particular item, returns its value
# Output: 16
print(squares.pop(4))

# Output: {1: 1, 2: 4, 3: 9, 5: 25}
print(squares)

# remove an arbitrary item, return (key,value)
# Output: (5, 25)
print(squares.popitem())

# Output: {1: 1, 2: 4, 3: 9}
print(squares)

# remove all items
squares.clear()

# Output: {}
print(squares)

# delete the dictionary itself
del squares

# Throws Error
print(squares)
```

Output

```
16
{1: 1, 2: 4, 3: 9, 5: 25}
(5, 25)
{1: 1, 2: 4, 3: 9}
{}
Traceback (most recent call last):
```

```
  File "<string>", line 30, in <module>
    print(squares)
NameError: name 'squares' is not defined
```

some more example:

```
# Dictionary Methods
marks = {}.fromkeys(['Math', 'English', 'Science'], 0)

# Output: {'English': 0, 'Math': 0, 'Science': 0}
print(marks)

for item in marks.items():
    print(item)

# Output: ['English', 'Math', 'Science']
print(list(sorted(marks.keys())))
```

Python Dictionary Comprehension

Dictionary comprehension is an elegant and concise way to create a new dictionary from an iterable in Python.

Dictionary comprehension consists of an expression pair (key: value) followed by a for statement inside curly braces {}.

Here is an example to make a dictionary with each item being a pair of a number and its square.

```
# Dictionary Comprehension
squares = {x: x*x for x in range(6)}

print(squares)
```

Output

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

This code is equivalent to

```
squares = {}
for x in range(6):
    squares[x] = x*x
print(squares)
```

Dictionary Membership Test

We can test if a key is in a dictionary or not using the keyword in. Notice that the membership test is only for the keys and not for the values.

```
# Membership Test for Dictionary Keys
squares = {1: 1, 3: 9, 5: 25, 7: 49, 9: 81}

# Output: True
print(1 in squares)

# Output: True
print(2 not in squares)

# membership tests for key only not value
# Output: False
print(49 in squares)
```

Iterating Through a Dictionary

We can iterate through each key in a dictionary using a for loop.

```
# Iterating through a Dictionary
squares = {1: 1, 3: 9, 5: 25, 7: 49, 9: 81}
for i in squares:
    print(squares[i])
```

Important dictionary methods:

Python Dictionary items()

The items() method returns a view object that displays a list of dictionary's (key, value) tuple pairs.

Example 1: Get all items of a dictionary with items()

```
# random sales dictionary
sales = { 'apple': 2, 'orange': 3, 'grapes': 4 }

print(sales.items())
```

Output

```
dict_items([('apple', 2), ('orange', 3), ('grapes', 4)])
```

Example 2: How items() works when a dictionary is modified?

```
# random sales dictionary
```

```
sales = { 'apple': 2, 'orange': 3, 'grapes': 4 }

items = sales.items()
print('Original items:', items)

# delete an item from dictionary
del[sales['apple']]
print('Updated items:', items)
```

Output

```
Original items: dict_items([('apple', 2), ('orange', 3), ('grapes', 4)])
Updated items: dict_items([('orange', 3), ('grapes', 4)])
```

The view object items doesn't itself return a list of sales items but it returns a view of sales's (key, value) pair.

If the list is updated at any time, the changes are reflected on to the view object itself, as shown in the above program.

Python Dictionary values()

The values() method returns a view object that displays a list of all the values in the dictionary.

The values() method returns a view object that displays a list of all values in a given dictionary.

```
# random sales dictionary
sales = { 'apple': 2, 'orange': 3, 'grapes': 4 }

print(sales.values())
```

Python Dictionary keys()

The keys() method returns a view object that displays a list of all the keys in the dictionary
The keys() returns a view object that displays a list of all the keys.

When the dictionary is changed, the view object also reflect these changes.

```
person = {'name': 'Phill', 'age': 22, 'salary': 3500.0}
print(person.keys())

empty_dict = {}
print(empty_dict.keys())
```

Example 2: How keys() works when dictionary is updated?

```
person = {'name': 'Phill', 'age': 22, }

print('Before dictionary is updated')
keys = person.keys()
print(keys)

# adding an element to the dictionary
person.update({'salary': 3500.0})
print('\nAfter dictionary is updated')
print(keys)
Output
Before dictionary is updated
dict_keys(['name', 'age'])

After dictionary is updated
dict_keys(['name', 'age', 'salary'])
```

Python if else

```
if condition :
        statement
else:
        statement
```

example palindrome program

```
def isPalindrome(string):
   left,right=0,len(string)-1  # here 0 assigned to left and strlen to right.

   while right>=left:
      if not string[left]==string[right]:
         return False
      left+=1;
      right-=1
      return True


if isPalindrome('redrum murder'):
   print ('palindrum')
else:
   print ('Not')
```

Well, there are other ways to do this too. Let's try using an iterator.

```
>>> def isPalindrome(string):
```

```
    left,right=iter(string),iter(string[::-1])
    i=0
    while i<len(string)/2:
        if next(left)!=next(right):
            return False
        i+=1
        return True
>>> isPalindrome('redrum murder')
```

What is meaning of left,right=iter(string),iter(string[::-1])

Here left is the iterator object of string, pointing to beginning and then other one right is pointing end as String[::-1] means return iterator of last element.

Python iter()

The Python iter() function returns an iterator for the given object.

Return value from iter(). The iter() function returns an iterator object for the given object.

If the user-defined object doesn't implement __iter__(), and __next__() or __getitem()__, the TypeError exception is raised.

If the sentinel parameter is also provided, iter() returns an iterator until the sentinel character isn't found.

Iterators are everywhere in Python. They are elegantly implemented within for loops, comprehensions, generators etc. but are hidden in plain sight.Iterator in Python is simply an object that can be iterated upon. An object which will return data, one element at a time.

Technically speaking, a Python iterator object must implement two special methods, __iter__() and __next__(), collectively called the iterator protocol.

An object is called iterable if we can get an iterator from it. Most built-in containers in Python like: list, tuple, string etc. are iterables.

The iter() function (which in turn calls the __iter__() method) returns an iterator from them.

```
# define a list
my_list = [4, 7, 0, 3]

# get an iterator using iter()
my_iter = iter(my_list)

# iterate through it using next()

# Output: 4
print(next(my_iter))

# Output: 7
print(next(my_iter))
```

```
# next(obj) is same as obj.__next__()

# Output: 0
print(my_iter.__next__())

# Output: 3
print(my_iter.__next__())

# This will raise error, no items left
next(my_iter)
```

Output

```
4
7
0
3
Traceback (most recent call last):
  File "<string>", line 24, in <module>
    next(my_iter)
StopIteration
```

Example 2: Working of Python iter()

```
# list of vowels
vowels = ['a', 'e', 'i', 'o', 'u']
vowels_iter = iter(vowels)

print(next(vowels_iter))    # 'a'
print(next(vowels_iter))    # 'e'
print(next(vowels_iter))    # 'i'
print(next(vowels_iter))    # 'o'
print(next(vowels_iter))    # 'u'
```

Output

```
a
e
i
o
u
```

Building Custom Iterators

Building an iterator from scratch is easy in Python. We just have to implement the __iter__() and the __next__() methods.

The __iter__() method returns the iterator object itself. If required, some initialization can be performed.
The __next__() method must return the next item in the sequence. On reaching the end, and in subsequent calls, it must raise StopIteration.
Here, we show an example that will give us the next power of 2 in each iteration. Power exponent starts from zero up to a user set number.

```
class PowTwo:
   """Class to implement an iterator
   of powers of two"""

   def __init__(self, max=0):
      self.max = max

   def __iter__(self):
      self.n = 0
      return self

   def __next__(self):
      if self.n <= self.max:
         result = 2 ** self.n
         self.n += 1
         return result
      else:
         raise StopIteration


# create an object
numbers = PowTwo(3)

# create an iterable from the object
i = iter(numbers)

# Using next to get to the next iterator element
print(next(i))
print(next(i))
print(next(i))
print(next(i))
print(next(i))
```

Output

```
1
2
4
8
Traceback (most recent call last):
  File "/home/bsoyuj/Desktop/Untitled-1.py", line 32, in <module>
    print(next(i))
  File "<string>", line 18, in __next__
    raise StopIteration
StopIteration
```

Python Functions

Example of a function

```
def greet(name):
    """
    This function greets to
    the person passed in as
    a parameter
    """
    print("Hello, " + name + ". Good morning!")

greet('Paul')
```

Docstrings

The first string after the function header is called the docstring and is short for documentation string. It is briefly used to explain what a function does.
or example:
Try running the following into the Python shell to see the output.

```
>>> print(greet.__doc__)

    This function greets to
    the person passed in as
    a parameter
```

Example of return

```
def absolute_value(num):
    """This function returns the absolute
    value of the entered number"""
```

```
    if num >= 0:
        return num
    else:
        return -num

print(absolute_value(2))
print(absolute_value(-4))
```

Scope and Lifetime of variables

Scope of a variable is the portion of a program where the variable is recognized. Parameters and variables defined inside a function are not visible from outside the function. Hence, they have a local scope.The lifetime of a variable is the period throughout which the variable exits in the memory. The lifetime of variables inside a function is as long as the function executes. They are destroyed once we return from the function. Hence, a function does not remember the value of a variable from its previous calls. Here is an example to illustrate the scope of a variable inside a function.

```
def my_func():
        x = 10
        print("Value inside function:",x)

x = 20
my_func()
print("Value outside function:",x)
```

Python Default Arguments

Function arguments can have default values in Python.
We can provide a default value to an argument by using the assignment operator (=). Here is an example.

```
def greet(name, msg="Good morning!"):
    """
    This function greets to
    the person with the
    provided message.

    If the message is not provided,
    it defaults to "Good
    morning!"
    """
    print("Hello", name + ', ' + msg)

greet("Kate")
```

```
greet("Bruce", "How do you do?")
```

Python allows functions to be called using keyword arguments. When we call functions in this way, the order (position) of the arguments can be changed. Following calls to the above function are all valid and produce the same result.

```
# 2 keyword arguments
greet(name = "Bruce",msg = "How do you do?")

# 2 keyword arguments (out of order)
greet(msg = "How do you do?",name = "Bruce")

1 positional, 1 keyword argument
greet("Bruce", msg = "How do you do?")
```

Python Arbitrary Arguments

Sometimes, we do not know in advance the number of arguments that will be passed into a function. Python allows us to handle this kind of situation through function calls with an arbitrary number of arguments.

In the function definition, we use an asterisk (*) before the parameter name to denote this kind of argument. Here is an example.

```
def greet(*names):
   """This function greets all
   the person in the names tuple."""

   # names is a tuple with arguments
   for name in names:
      print("Hello", name)


greet("Monica", "Luke", "Steve", "John")
```

Python Anonymous/Lambda Function

Example of Lambda Function in python. Here is an example of lambda function that doubles the input value.

```
# Program to show the use of lambda functions
double = lambda x: x * 2
```

```
print(double(5))
```

Example use with filter()
The filter() function in Python takes in a function and a list as arguments.
The function is called with all the items in the list and a new list is returned which contains items for which the function evaluates to True.

Here is an example use of filter() function to filter out only even numbers from a list.

```
# Program to filter out only the even items from a list
my_list = [1, 5, 4, 6, 8, 11, 3, 12]
new_list = list(filter(lambda x: (x%2 == 0) , my_list))
print(new_list)
```

Example use with map()
The map() function in Python takes in a function and a list.
The function is called with all the items in the list and a new list is returned which contains items returned by that function for each item. Here is an example use of map() function to double all the items in a list.

```
# Program to double each item in a list using map()

my_list = [1, 5, 4, 6, 8, 11, 3, 12]
new_list = list(map(lambda x: x * 2 , my_list))
print(new_list)
```

Python Global, Local and Nonlocal variables

Global Variables

In Python, a variable declared outside of the function or in global scope is known as a global variable. This means that a global variable can be accessed inside or outside of the function.

Let's see an example of how a global variable is created in Python.

Example 1: Create a Global Variable

```
x = "global"

def foo():
    print("x inside:", x)
```

```
foo()
print("x outside:", x)
```

output:

```
x inside: global
x outside: global
```

What if you want to change the value of x inside a function?

```
x = "global"

def foo():
    x = x * 2
    print(x)

foo()
```

Output

UnboundLocalError: local variable 'x' referenced before assignment
The output shows an error because Python treats x as a local variable and x is also not defined inside foo().

solution:
To make this work, we use the global keyword.

Local variable:

Example 2: Accessing local variable outside the scope

```
def foo():
    y = "local"


foo()
print(y)
```

Run Code
Output

```
NameError: name 'y' is not defined
```

Global and local variables

Example : Using Global and Local variables in the same code

```
x = "global "

def foo():
    global x
    y = "local"
    x = x * 2
    print(x)
    print(y)

foo()
```

Run Code

Output

```
global global
local
```

In the above code, we declare x as a global and y as a local variable in the foo().
Then, we use multiplication operator * to modify the global variable x and we print both x and y.
After calling the foo(), the value of x becomes global global because we used the x * 2 to print two times global. After that, we print the value of local variable y i.e local.

Example : Global variable and Local variable with same name

```
x = 5

def foo():
    x = 10
    print("local x:", x)


foo()
print("global x:", x)
```

Output

```
local x: 10
global x: 5
```

Nonlocal Variables

Nonlocal variables are used in nested functions whose local scope is not defined. This means that the variable can be neither in the local nor the global scope. Let's see an example of how a global variable is created in Python.
We use nonlocal keywords to create nonlocal variables.

Example : Create a nonlocal variable

```
def outer():
   x = "local"

   def inner():
      nonlocal x
      x = "nonlocal"
      print("inner:", x)

   inner()
   print("outer:", x)


outer()
```

Output

```
inner: nonlocal
outer: nonlocal
```

In the above code, there is a nested inner() function. We use nonlocal keywords to create a nonlocal variable. The inner() function is defined in the scope of another function outer().

Note : If we change the value of a nonlocal variable, the changes appear in the local variable.

Understanding if __name__ == "__main__" in Python

Every module in Python has a special attribute called __name__. The value of __name__ attribute is set to "__main__" when module is run as main program. Otherwise, the value of __name__ is set to contain the name of the module. We use if __name__ == "__main__" block to prevent (certain) code from being run when the module is imported.

example:

Let's put together this little code example for understanding. Suppose we create two modules, foo and bar with the following code:

```
# foo.py
import bar
print("foo.__name__ set to ", __name__)
```

And the bar module:

```
# bar.py
print("bar.__name__ set to ", __name__)
```

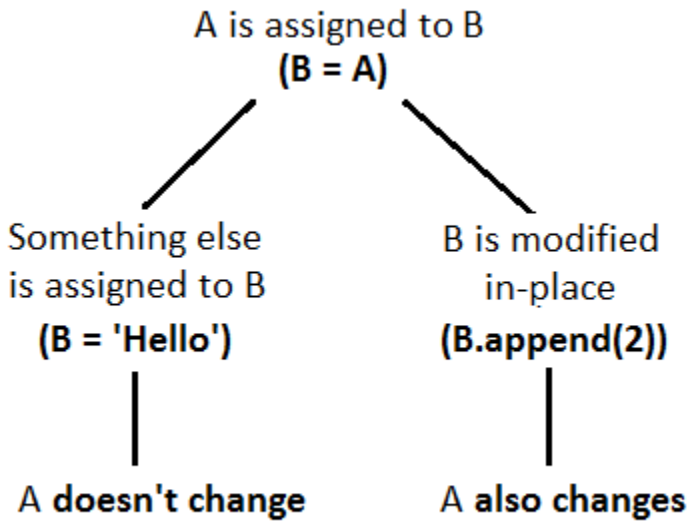
On invoking the bar module from command line, its __name__ attribute will be set to "__main__":

python bar.py
>>>
bar.__name__ set to __main__
However, on invoking the foo module in a similar fashion, the bar's __name__ attribute will be set equivalent to it's module name i.e bar:
python foo.py
>>>
bar.__name__ set to bar
foo.__name__ set to __main__

Is Python call by reference or call by value

B=A;

It doesn't matter if A is mutable or not. If you assign something different to B, A doesn't change.




```
a = [1, 2, 3]
b = a
b.append(4)
b = ['a', 'b']
print a, b      # prints [1, 2, 3, 4] ['a', 'b']
```

Reason is Assignment statements modify namespaces, not objects.

In other words,

name = 10

means that you're adding the name "name" to your local namespace, and making it refer to an integer object containing the value 10.

If the name is already present, the assignment replaces the original name:

name = 10
name = 20

means that you're first adding the name "name" to the local namespace, and making it refer to an integer object containing the value 10. You're then replacing the name, making it point to an integer object containing the value 20. The original "10" object isn't affected by this operation, and it doesn't care.
In contrast, if you do:

name = []
name.append(1)

you're first adding the name "name" to the local namespace, making it refer to an empty list object. This modifies the namespace. You're then calling a method on that object, telling it to append an integer object to itself. This modifies the content of the list object, but it doesn't touch the namespace, and it doesn't touch the integer object.

If we pass a list to a function, we have to consider two cases: Elements of a list can be changed in place, i.e. the list will be changed even in the caller's scope. If a new list is assigned to the name, the old list will not be affected, i.e. the list in the caller's scope will remain untouched.

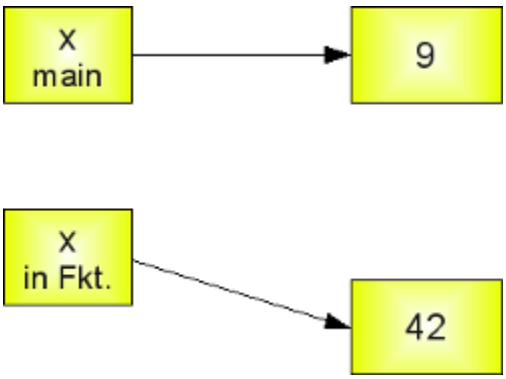
Example:

```
def ref_demo(x):
    print ("x=",x )
    print (id Of X = ",id(x))
    x=42
    print ("x=",x )
    print (id Of X = ",id(x))

x=9;
ref_demo(x)
print ("x=",x )
print (id Of X = ",id(x))
```

OUTPUT

```
x= 9
id Of X =  140722809762320
x= 42
id Of X =  140722809763376
x= 9
id Of X =  140722809762320
```

This means that Python initially behaves like call-by-reference, but as soon as we are changing the value of such a variable, Python "switches" to call-by-value.



What is Name in Python?

Name (also called identifier) is simply a name given to objects. Everything in Python is an object. Name is a way to access the underlying object.

For example, when we do the assignment a = 2, 2 is an object stored in memory and a is the name we associate it with.

We can get the address (in RAM) of some object through the built-in function id(). Let's look at how to use it.

```
a = 2
print('id(2) =', id(2))
print('id(a) =', id(a))
```

Output

```
id(2) = 9302208
id(a) = 9302208
```

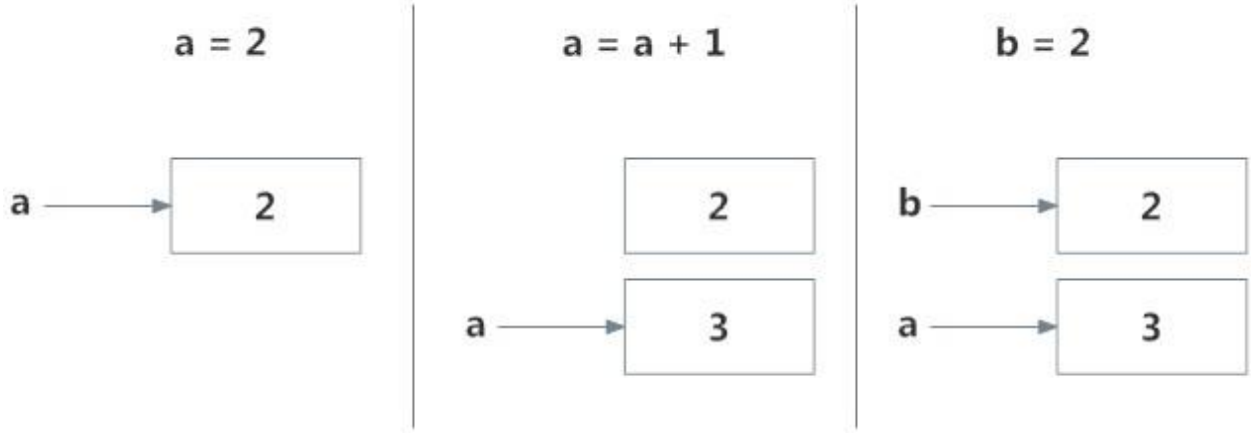
Here, both refer to the same object 2, so they have the same id(). Let's make things a little more interesting.

```
# Note: You may get different values for the id
a = 2
print('id(a) =', id(a))
a = a+1
print('id(a) =', id(a))
print('id(3) =', id(3))
b = 2
print('id(b) =', id(b))
print('id(2) =', id(2))
```

Output

```
id(a) = 9302208
id(a) = 9302240
id(3) = 9302240
id(b) = 9302208
```

id(2) = 9302208
What is happening in the above sequence of steps? Let's use a diagram to explain this:



Memory diagram of variables in Python

Initially, an object 2 is created and the name a is associated with it, when we do a = a+1, a new object 3 is created and now a is associated with this object.
Note that id(a) and id(3) have the same values.
Furthermore, when b = 2 is executed, the new name b gets associated with the previous object 2. This is efficient as Python does not have to create a new duplicate object. This dynamic nature of name binding makes Python powerful;
a name could refer to any type of object.

```
>>> a = 5
>>> a = 'Hello World!'
>>> a = [1,2,3]
```

All these are valid and a will refer to three different types of objects in different instances.
Functions are objects too, so a name can refer to them as well.

```
def printHello():
    print("Hello")
a = printHello

a()
```

Output

Hello
The same name a can refer to a function and we can call the function using this name.

Python Modules

Modules refer to a file containing Python statements and definitions.
A file containing Python code, for example: example.py, is called a module, and its module name would be example.Let us create a module. Type the following and save it as example.py.

 # Python Module example

```
def add(a, b):
   """This program adds two
   numbers and return the result"""

   result = a + b
   return result
```

Here, we have defined a function add() inside a module named example. The function takes in two numbers and returns their sum.

How to import modules in Python?

We can import the definitions inside a module to another module or the interactive interpreter in Python.

We use the import keyword to do this. To import our previously defined module example, we type the following in the Python prompt.

```
>>> import example
```

Using the module name we can access the function using the dot . operator. For example:

```
>>> example.add(4,5.5)
9.5
```

Reloading a module

We can use the reload() function inside the imp module to reload a module. We can do it in the following ways:

```
>>> import imp
```

```
>>> import my_module
This code got executed
>>> import my_module
>>> imp.reload(my_module)
This code got executed
<module 'my_module' from '.\\my_module.py'>
```

Python Module Search Path

While importing a module, Python looks at several places. Interpreter first looks for a built-in module. Then(if built-in module not found), Python looks into a list of directories defined in sys.path. The search is in this order.

The current directory.

PYTHONPATH (an environment variable with a list of directories).
The installation-dependent default directory.

```
>>> import sys
>>> sys.path
['',
'C:\\Python33\\Lib\\idlelib',
'C:\\Windows\\system32\\python33.zip',
'C:\\Python33\\DLLs',
'C:\\Python33\\lib',
'C:\\Python33',
'C:\\Python33\\lib\\site-packages']
```

We can add and modify this list to add our own path.

The dir() built-in function

We can use the dir() function to find out names that are defined inside a module.

For example, we have defined a function add() in the module example that we had in the beginning.

We can use dir in example module in the following way:

```
>>> dir(example)
['__builtins__',
'__cached__',
'__doc__',
'__file__',
```

'__initializing__',
'__loader__',
'__name__',
'__package__',
'add']

Here, we can see a sorted list of names (along with add). All other names that begin with an underscore are default Python attributes associated with the module (not user-defined).

For example, the __name__ attribute contains the name of the module.

```
>>> import example
>>> example.__name__
'example'
```

What is a Namespace in Python?

Now that we understand what names are, we can move on to the concept of namespaces.
To simply put it, a namespace is a collection of names.
In Python, you can imagine a namespace as a mapping of every name you have defined to corresponding objects.
Different namespaces can co-exist at a given time but are completely isolated.

A namespace containing all the built-in names is created when we start the Python interpreter and exists as long as the interpreter runs.
This is the reason that built-in functions like id(), print() etc. are always available to us from any part of the program. Each module creates its own global namespace.
These different namespaces are isolated. Hence, the same name that may exist in different modules do not collide.

Modules can have various functions and classes. A local namespace is created when a function is called,
which has all the names defined in it. Similar, is the case with class. Following diagram may help to clarify this concept.

When you call a function, a local python namespace is created for all the names in it. A module has a global namespace. The built-in namespace encloses this. Take a look at the following figure to get a clearer understanding.

A namespace in python is a collection of names. So, a namespace is essentially a mapping of names to corresponding objects. At any instant, different python namespaces can coexist completely isolated- the isolation ensures that there are no name collisions. Simply speaking, two namespaces in python can have the same name without facing any problem. A namespace is implemented as a Python dictionary.
Python assumes that any variable assigned a value in a function is local.
Therefore, in order to assign a value to a global variable within a function, you must first use the global statement.
When we start the interpreter, a python namespace is created for as long as we don't exist. This holds all built-in names. It is due to this that python functions like print () and id () are always available. Also, each module creates its own global namespace in python.

At any instant, we have at least three nested python scopes:

- Current function's variable scope- has local names
- Module's variable scope- has global names
- The outermost variable scope- has built-in names

Example of Scope and Namespace in Python

```
def outer_function():
    b = 20
    def inner_func():
        c = 30

a = 10
```

Here, the variable a is in the global namespace.
Variable b is in the local namespace of outer_function() and c is in the nested local namespace of inner_function().

What is the global keyword?

In Python, global keyword allows you to modify the variable outside of the current scope. It is used to create a global variable and make changes to the variable in a local context.

Rules of global Keyword

- The basic rules for global keyword in Python are:
- When we create a variable inside a function, it is local by default.
- When we define a variable outside of a function, it is global by default. You don't have to use global keyword.
- We use global keyword to read and write a global variable inside a function.
- Use of global keyword outside a function has no effect.

Example 1: Modifying Global Variable from Inside the Function

```
c = 1 # global variable
def add():
    c = c + 2 # increment c by 2
    print(c)
add()
```

When we run the above program, the output shows an error:

UnboundLocalError: local variable 'c' referenced before assignment

This is because we can only access the global variable but cannot modify it from inside the function.

The solution for this is to use the global keyword.

Example 2: Changing Global Variable From Inside a Function using global

```
c = 0 # global variable

def add():
    global c
    c = c + 2 # increment by 2
    print("Inside add():", c)

add()
print("In main:", c)
```

Example 4: Share a global Variable Across Python Modules

Create a **config.py** file, to store global variables

config.py

```
a = 0
```

b = "empty"
Create a update.py file, to change global variables

update.py

```
import config
config.a = 10
config.b = "alphabet"
```

Create a main.py file, to test changes in value

main.py

```
import config
import update

print(config.a)
print(config.b)
```

When we run the main.py file, the output will be

```
10
Alphabet
```

In the above, we have created three files: config.py, update.py, and main.py.

The module config.py stores global variables of a and b. In the update.py file, we import the config.py module and modify the values of a and b.
Similarly, in the main.py file, we import both config.py and update.py module. Finally, we print and test the values of global variables
whether they are changed or not.
Global in Nested Functions
Here is how you can use a global variable in nested function.

Example 5: Using a Global Variable in Nested Function

```
def foo():
    x = 20

    def bar():
        global x
        x = 25

    print("Before calling bar: ", x)
    print("Calling bar now")
    bar()
    print("After calling bar: ", x)

foo()
```

```
print("x in main: ", x)
```

```
Before calling bar: 20
Calling bar now
After calling bar: 20
x in main: 25
```

In the above program, we declared a global variable inside the nested function bar().
Inside foo() function, x has no effect of the global keyword.
Before and after calling bar(), the variable x takes the value of local variable i.e x = 20. Outside of the foo() function,
the variable x will take value defined in the bar() function i.e x = 25.
This is because we have used global keyword in x to create global variable inside the bar() function (local scope).
If we make any changes inside the bar() function, the changes appear outside the local scope, i.e. foo().

Python File and I/O:

Python Files:

open() returns a file object, and is most commonly used with two arguments: open(filename, mode).

" r ", for reading.
" w ", for writing.
" a ", for appending.
" r+ ", for both reading and writing

```
f = open('workfile', 'w')
```

Unlike other languages, the character a does not imply the number 97 until it is encoded using ASCII (or other equivalent encodings).
Moreover, the default encoding is platform dependent. In windows, it is cp1252 but utf-8 in Linux.
So, we must not also rely on the default encoding or else our code will behave differently in different platforms.
Hence, when working with files in text mode, it is highly recommended to specify the encoding type.

```
f = open("test.txt", mode='r', encoding='utf-8')
```

A safer way is to use a try...finally block.

```
try:
  f = open("test.txt", encoding = 'utf-8')
```

```
  # perform file operations
finally:
  f.close()
```

But, it is good practice to use the with keyword when dealing with file objects. The advantage is that the file is properly closed after its suite finishes,
even if an exception is raised at some point. Using with is also much shorter than writing equivalent try-finally blocks:

```
with open("test.txt", encoding = 'utf-8') as f:
  # perform file operations
```

More example:

```
with open('dog_breeds.txt', 'r') as reader:
>>>    # Read and print the entire file line by line
>>>    line = reader.readline()
>>>    while line != '':  # The EOF char is an empty string
>>>        print(line, end='')
>>>        line = reader.readline()
```

However, the above examples can be further simplified by iterating over the file object itself:

```
>>> with open('dog_breeds.txt', 'r') as reader:
>>>    # Read and print the entire file line by line
>>>    for line in reader:
>>>        print(line, end='')
```

This final approach is more Pythonic and can be quicker and more memory efficient. Therefore, it is suggested you use this instead.

Note: Some of the above examples contain print('some text', end='').
The end='' is to prevent Python from adding an additional newline to the text that is being printed and only print what is being read from the file.

Here's a quick example of using .write() and .writelines():

```
with open('dog_breeds.txt', 'r') as reader:
   # Note: readlines doesn't trim the line endings
   dog_breeds = reader.readlines()

with open('dog_breeds_reversed.txt', 'w') as writer:
   # Alternatively you could use
```

```
    # writer.writelines(reversed(dog_breeds))

    # Write the dog breeds to the file in reversed order
    for breed in reversed(dog_breeds):
        writer.write(breed)
```

Reading Files in Python

```
>>> with open('dog_breeds.txt', 'r') as reader:
>>>     # Read & print the entire file
>>>     print(reader.read())
```

Here's an example of how to read the entire file as a list using the Python .readlines() method:

```
>>> f = open('dog_breeds.txt')
>>> f.readlines()  # Returns a list object
['Pug\n', 'Jack Russell Terrier\n', 'English Springer Spaniel\n', 'German Shepherd\n', 'Staffordshire Bull Terrier\n',
'Cavalier King Charles Spaniel\n', 'Golden Retriever\n', 'West Highland White Terrier\n', 'Boxer\n',
'Border Terrier\n']
```

The above example can also be done by using list() to create a list out of the file object:

```
>>> f = open('dog_breeds.txt')
>>> list(f)
['Pug\n', 'Jack Russell Terrier\n', 'English Springer Spaniel\n', 'German Shepherd\n
```

Iterating Over Each Line in the File

```
>>> with open('dog_breeds.txt', 'r') as reader:
>>>     # Read and print the entire file line by line
>>>     line = reader.readline()
>>>     while line != '':  # The EOF char is an empty string
>>>         print(line, end='')
>>>         line = reader.readline()
```

Example 1: Read CSV Having Comma Delimiter

```
import csv
with open('people.csv', 'r') as file:
    reader = csv.reader(file)
    for row in reader:
```

```
        print(row)
```
Output

```
['Name', 'Age', 'Profession']
['Jack', '23', 'Doctor']
['Miller', '22', 'Engineer']
```

Example 2: Read CSV file Having Tab Delimiter

```
import csv
with open('people.csv', 'r',) as file:
   reader = csv.reader(file, delimiter = '\t')
   for row in reader:
      print(row)
```

Example 3: Write to a CSV file

```
import csv
with open('protagonist.csv', 'w', newline='') as file:
   writer = csv.writer(file)
   writer.writerow(["SN", "Movie", "Protagonist"])
   writer.writerow([1, "Lord of the Rings", "Frodo Baggins"])
   writer.writerow([2, "Harry Potter", "Harry Potter"])
```

Python Lambda:

The general syntax of a lambda function is quite simple:
lambda argument_list: expression
The following example of a lambda function returns the sum of its two arguments:

```
sum = lambda x, y : x + y
sum(3,4)
Output::
7
```

Same can be done with Function:

```
def sum(x,y):
   return x + y

sum(3,4)
Output::
7
```

The map() Function

As we have mentioned earlier, the advantage of the lambda operator can be seen when it is used in combination with the map() function. map() is a function which takes two arguments:

r = map(func, seq)

The built-in function map() takes a function as a first argument and applies it to each of the elements of its second argument, an iterable.
Examples of iterables are strings, lists, and tuples. For more information on iterables and iterators, check out Iterables and Iterators.

map() returns an **iterator** corresponding to the transformed collection. As an example, if you wanted to transform a list of strings to a new list
with each string capitalized, you could use map(), as follows:

```
>>> list(map(lambda x: x.capitalize(), ['cat', 'dog', 'cow']))
['Cat', 'Dog', 'Cow']
```

You need to invoke list() to convert the iterator returned by map() into an expanded list that can be displayed in the Python shell interpreter.

Using a list comprehension eliminates the need for defining and invoking the lambda function:

```
>>> [x.capitalize() for x in ['cat', 'dog', 'cow']]
['Cat', 'Dog', 'Cow']
```

The following example illustrates the way of working of map():

```
def fahrenheit(T):
   return ((float(9)/5)*T + 32)

def celsius(T):
   return (float(5)/9)*(T-32)

temperatures = (36.5, 37, 37.5, 38, 39)
F = map(fahrenheit, temperatures)
C = map(celsius, F)
temperatures_in_Fahrenheit = list(map(fahrenheit, temperatures))
temperatures_in_Celsius = list(map(celsius, temperatures_in_Fahrenheit))
print(temperatures_in_Fahrenheit)
print(temperatures_in_Celsius)
```

Output:

[97.7, 98.60000000000001, 99.5, 100.4, 102.2]
[36.5, 37.00000000000001, 37.5, 38.00000000000001, 39.0]

Same with lambda:

```
C = [39.2, 36.5, 37.3, 38, 37.8]
F = list(map(lambda x: (float(9)/5)*x + 32, C))
print(F)
[102.56, 97.7, 99.14, 100.4, 100.03999999999999]
C = list(map(lambda x: (float(5)/9)*(x-32), F))
print(C)
```

Python Filter

The built-in function filter (), another classic functional construct, can be converted into a list comprehension. It takes a predicate as a first argument and an iterable as a second argument. It builds an iterator containing all the elements of the initial collection that satisfies the predicate function. Here's an example that filters all the even numbers in a given list of integers:

```
>>> even = lambda x: x%2 == 0
>>> list (filter (even, range (11)))
[0, 2, 4, 6, 8, 10]
```

Note that filter () returns an iterator, hence the need to invoke the built-in type list that constructs a list given an iterator.

The implementation leveraging the list comprehension construct gives the following:

```
>>> [x for x in range (11) if x%2 == 0]
[0, 2, 4, 6, 8, 10]
```

```
mylist = [0,1,1,2,3,5,8,13,21,34,55]
even_numbers = list (filter (lambda x: x % 2 == 0, mylist))
print(even_numbers)
[0, 2, 8, 34]
```

Python Directory and Files Management

Get Current Directory

```
import os
os.getcwd()
os.getcwdb()
```

Changing Directory

os.chdir('C:\\Python33')

List Directories and Files

os.listdir()

Making a New Directory

os.mkdir('test')

Python Built-in Exceptions

We can make certain mistakes while writing a program that lead to errors when we try to run it. A python program terminates as soon as
it encounters an unhandled error. These errors can be broadly classified into two classes:

- Syntax errors
- Logical errors (Exceptions)

devide by zero is a logical error.

We can view all the built-in exceptions using the built-in local() function as follows:

print(dir(locals()['__builtins__'])) → **if u run at cmd prompt, you will get below stuffs.**

['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'BlockingIOError',
'BrokenPipeError', 'BufferError', 'BytesWarning',
'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError', 'ConnectionRefusedError',
'ConnectionResetError', 'DeprecationWarning',
'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False', 'FileExistsError', 'FileNotFoundError',
'FloatingPointError', 'FutureWarning',
'GeneratorExit', 'IOError', 'ImportError', 'ImportWarning', 'IndentationError', 'IndexError',
'InterruptedError', 'IsADirectoryError',
'KeyError', 'KeyboardInterrupt', 'LookupError', 'MemoryError', 'ModuleNotFoundError', 'NameError',
'None', 'NotADirectoryError', 'NotImplemented',
'NotImplementedError', 'OSError', 'OverflowError', 'PendingDeprecationWarning', 'PermissionError',
'ProcessLookupError', 'RecursionError',
'ReferenceError', 'ResourceWarning', 'RuntimeError', 'RuntimeWarning', 'StopAsyncIteration',
'StopIteration', 'SyntaxError', 'SyntaxWarning',
'SystemError', 'SystemExit', 'TabError', 'TimeoutError', 'True', 'TypeError', 'UnboundLocalError',
'UnicodeDecodeError', 'UnicodeEncodeError',
'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning', 'ValueError', 'Warning',
'WindowsError', 'ZeroDivisionError',
'__build_class__', '__debug__', '__doc__', '__import__', '__loader__', '__name__', '__package__',
'__spec__', 'abs', 'all', 'any', 'ascii',
'bin', 'bool', 'breakpoint', 'bytearray', 'bytes', 'callable', 'chr', 'classmethod', 'compile', 'complex',
'copyright', 'credits', 'delattr',

'dict', 'dir', 'divmod', 'enumerate', 'eval', 'exec', 'exit', 'filter', 'float', 'format', 'frozenset', 'getattr', 'globals', 'hasattr',
'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance', 'issubclass', 'iter', 'len', 'license', 'list', 'locals', 'map', 'max', 'memoryview',
'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property', 'quit', 'range', 'repr', 'reversed', 'round', 'set', 'setattr',
'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']

Exceptions in Python

Python has many built-in exceptions that are raised when your program encounters an error (something in the program goes wrong).
When these exceptions occur, the Python interpreter stops the current process and passes it to the calling process until it is handled.
If not handled, the program will crash.
For example, let us consider a program where we have a function A that calls function B, which in turn calls function C.
If an exception occurs in function C but is not handled in C, the exception passes to B and then to A.

If never handled, an error message is displayed and our program comes to a sudden unexpected halt.

Catching Exceptions in Python

If you run at cmd prompt, then supply other then int, you will get below error:

```
n = int(input("Please enter a number: "))
Please enter a number: 23.5
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-9-02fbe8840e1a> in <module>
----> 1 n = int(input("Please enter a number: "))

ValueError: invalid literal for int() with base 10: '23.5'
```

Now we will handle the exception.

```
while True:
    try:
        n = input("Please enter an integer: ")
        n = int(n)
        break
    except ValueError:
```

```
        print("No valid integer! Please try again ...")
print("Great, you successfully entered an integer!")
```

Catching Specific Exceptions in Python:

```
import sys

try:
   f = open('integers.txt')
   s = f.readline()
   i = int(s.strip())
except IOError as e:
   errno, strerror = e.args
   print("I/O error({0}): {1}".format(errno,strerror))
   # e can be printed directly without using .args:
   # print(e)
except ValueError:
   print("No valid integer in line.")
except:
   print("Unexpected error:", sys.exc_info()[0])
   raise
```

Raising Exceptions in Python

In Python programming, exceptions are raised when errors occur at runtime. We can also manually raise exceptions using the raise keyword.
We can optionally pass values to the exception to clarify why that exception was raised.

```
>>> raise KeyboardInterrupt
Traceback (most recent call last):
...
KeyboardInterrupt

>>> raise MemoryError("This is an argument")
Traceback (most recent call last):
...
MemoryError: This is an argument

>>> try:
...     a = int(input("Enter a positive integer: "))
...     if a <= 0:
...         raise ValueError("That is not a positive number!")
... except ValueError as ve:
...     print(ve)
```

...
Enter a positive integer: -2
That is not a positive number!

Combining try, except and finally

"finally" and "except" can be used together for the same try block, as it can be seen in the following Python example:
The try statement in Python can have an optional finally clause. This clause is executed no matter what, and is generally used to release external resources.

For example, we may be connected to a remote data center through the network or working with a file or a Graphical User Interface (GUI). In all these circumstances, we must clean up the resource before the program comes to a halt whether it successfully ran or not. These actions (closing a file, GUI or disconnecting from network) are performed in the finally clause to guarantee the execution.

```
try:
   x = float(input("Your number: "))
   inverse = 1.0 / x
except ValueError:
   print("You should have given either an int or a float")
except ZeroDivisionError:
   print("Infinity")
finally:
   print("There may or may not have been an exception.")
```

```
Your number: 23
There may or may not have been an exception.
```

some more example:

```
try:
  f = open("test.txt",encoding = 'utf-8')
  # perform file operations
finally:
  f.close()
```

Python try with else clause

```
import sys
file_name = sys.argv[1]
text = []
try:
   fh = open(file_name, 'r')
except IOError:
```

```
    print('cannot open', file_name)
else:
    text = fh.readlines()
    fh.close()

if text:
    print(text[100])
```

Python Custom Exceptions

Example: User-Defined Exception in Python

In this example, we will illustrate how user-defined exceptions can be used in a program to raise and catch errors.This program will ask the user to enter a number until they guess a stored number correctly. To help them figure it out, a hint is provided whether their guess is greater than or less than the stored number.

```
# define Python user-defined exceptions
class Error(Exception):
    """Base class for other exceptions"""
    pass

class ValueTooSmallError(Error):
    """Raised when the input value is too small"""
    pass

class ValueTooLargeError(Error):
    """Raised when the input value is too large"""
    pass

# you need to guess this number
number = 10

# user guesses a number until he/she gets it right
while True:
    try:
        i_num = int(input("Enter a number: "))
        if i_num < number:
            raise ValueTooSmallError
        elif i_num > number:
            raise ValueTooLargeError
        break
    except ValueTooSmallError:
        print("This value is too small, try again!")
```

```
        print()
    except ValueTooLargeError:
        print("This value is too large, try again!")
        print()

print("Congratulations! You guessed it correctly.")
```

Here is a sample run of this program.

```
Enter a number: 12
This value is too large, try again!
Enter a number: 0
This value is too small, try again!
Enter a number: 8
This value is too small, try again!
Enter a number: 10
Congratulations! You guessed it correctly.
```

Some more example of customized exception:

```
class SalaryNotInRangeError(Exception):
    """Exception raised for errors in the input salary.

    Attributes:
        salary -- input salary which caused the error
        message -- explanation of the error
    """

    def __init__(self, salary, message="Salary is not in (5000, 15000) range"):
        self.salary = salary
        self.message = message
        super().__init__(self.message)


salary = int(input("Enter salary amount: "))
if not 5000 < salary < 15000:
    raise SalaryNotInRangeError(salary)
```

Output

```
Enter salary amount: 2000
Traceback (most recent call last):
  File "<string>", line 17, in <module>
```

```
    raise SalaryNotInRangeError(salary)
__main__.SalaryNotInRangeError: Salary is not in (5000, 15000) range
```

Here, we have overridden the constructor of the Exception class to accept our own custom arguments salary and message.
Then, the constructor of the parent Exception class is called manually with the self.message argument using super().

The custom self.salary attribute is defined to be used later.

The inherited __str__ method of the Exception class is then used to display the corresponding message when SalaryNotInRangeError is raised.

We can also customize the __str__ method itself by overriding it.

```
class SalaryNotInRangeError(Exception):
    """Exception raised for errors in the input salary.

    Attributes:
        salary -- input salary which caused the error
        message -- explanation of the error
    """

    def __init__(self, salary, message="Salary is not in (5000, 15000) range"):
        self.salary = salary
        self.message = message
        super().__init__(self.message)

    def __str__(self):
        return f'{self.salary} -> {self.message}'

salary = int(input("Enter salary amount: "))
if not 5000 < salary < 15000:
    raise SalaryNotInRangeError(salary)
```

Output

```
Enter salary amount: 2000
Traceback (most recent call last):
  File "/home/bsoyuj/Desktop/Untitled-1.py", line 20, in <module>
    raise SalaryNotInRangeError(salary)
__main__.SalaryNotInRangeError: 2000 -> Salary is not in (5000, 15000) range
```

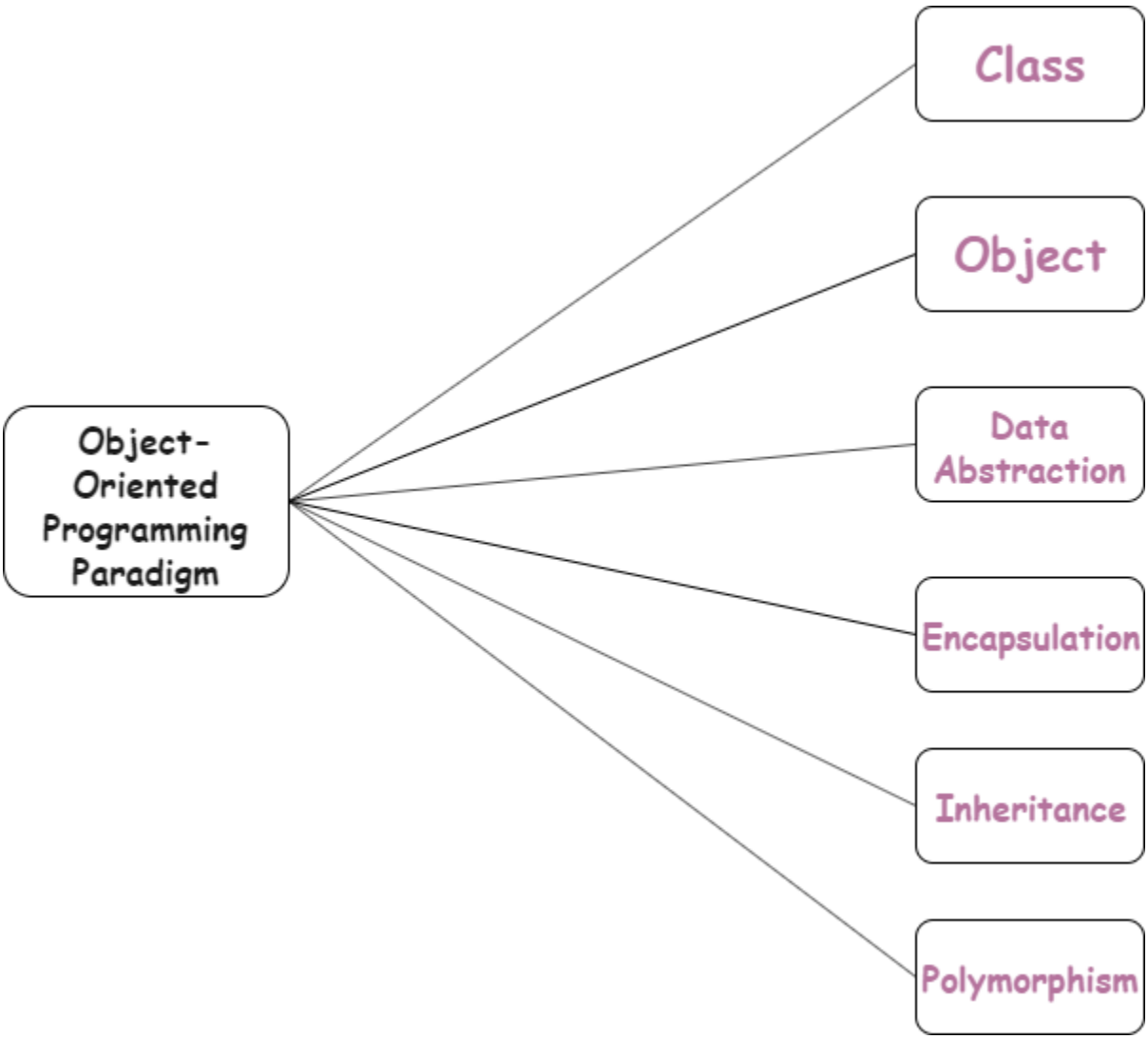
Python Object Oriented Programming

Object-oriented programming is a programming paradigm that provides a means of structuring programs so that properties and behaviors
are bundled into individual objects.
For instance, an object could represent a person with properties like a name, age, and address and behaviors such as walking, talking, breathing,
and running. Or it could represent an email with properties like a recipient list, subject, and body and behaviors like adding attachments and sending.

Another common programming paradigm is procedural programming, which structures a program like a recipe in that it provides a set of steps, in the form of functions and code blocks,
that flow sequentially in order to complete a task.

The key takeaway is that objects are at the center of object-oriented programming in Python, not only representing the data,
as in procedural programming, but in the overall structure of the program as well.

This paradigm maps and models real-world things together and describes a relationship among them. OOP models real-world entities as software objects, which have data associated with them and have some behavioral patterns (functions).



Define a Class in Python

How class solve below problem?

One way to do this is to represent each employee as a list:

kirk = ["James Kirk", 34, "Captain", 2265]

But this is bit cumbersome. A great way to make this type of code more manageable and more maintainable is to use classes.

Classes vs Instances

Classes are used to create user-defined data structures. Classes define functions called methods, which identify the behaviors and

actions that an object created from the class can perform with its data.

While the class is the blueprint, an instance is an object that is built from a class and contains real data. An instance of the Dog class is not a blueprint anymore. It's an actual dog with a name, like Miles, who's four years old.

An object has two characteristics:

- attributes
- behavior

How to Define a Class
Class
A class is a blueprint for the object.

```
class Parrot:
    pass
```

what is pass in python?

The pass statement is used as a placeholder for future code. When the pass statement is executed, nothing happens, but you avoid getting an error when empty code is not allowed. Empty code is not allowed in loops, function definitions, class definitions, or in if statements.

Object
An object (instance) is an instantiation of a class. When class is defined, only the description for the object is defined.
Therefore, no memory or storage is allocated.

The example for object of parrot class can be:

```
obj = Parrot()
```

Example 1: Creating Class and Object in Python

```
class Parrot:

    # class attribute  -> something like static variable in c++ or Java,which belongs to class not to object
    species = "bird"

    # instance attribute --> Something like constructor in c++/Java
    def __init__(self, name, age):
```

```
        self.name = name
        self.age = age

# instantiate the Parrot class
blu = Parrot("Blu", 10)
woo = Parrot("Woo", 15)

# access the class attributes
print("Blu is a {}".format(blu.__class__.species))
print("Woo is also a {}".format(woo.__class__.species))

# access the instance attributes
print("{} is {} years old".format( blu.name, blu.age))
print("{} is {} years old".format( woo.name, woo.age))
```

Output

```
Blu is a bird
Woo is also a bird
Blu is 10 years old
Woo is 15 years old
```

We can access the class attribute using __class__.species. Class attributes are the same for all instances of a class. Similarly,
we access the instance attributes using blu.name and blu.age. However, instance attributes are different for every instance of a class.

Example 2: One More example

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

   def __str__(self):
      print(self.x)
      print(self.y)
      return "({0},{1})".format(self.x, self.y)

p1 = Point(11,22)
print(p1.x,p1.y)
```

str(p1);

With this example, we get to know that self is not a keyword, we can declare anything like me etc.
In the first argument to __init__(), which is a python define method. just like __str__();
when you create an object, it pass the object to init (by default) as first argument. So you don't have to worry about it, you just pass the argument that your __init__() takes.

when you want to prin the object like print (p1), It will print the address of the object, if you have not defined the __str__().

Methods

Methods are functions defined inside the body of a class. They are used to define the behaviors of an object.

Example 3 : Creating Methods in Python

```
class Parrot:

    # instance attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # instance method
    def sing(self, song):
        return "{} sings {}".format(self.name, song)

    def dance(self):
        return "{} is now dancing".format(self.name)

# instantiate the object
blu = Parrot("Blu", 10)

# call our instance methods
print(blu.sing("'Happy'"))
print(blu.dance())
```

Python Doesn't Support Multiple Constructors

Python doesn't support multiple constructors, unlike other popular object-oriented programming languages such as Java.
We can define multiple __init__() methods but the last one will override the earlier definitions.

```
class D:
```

```
    def __init__(self, x):
        print(f'Constructor 1 with argument {x}')

    # this will overwrite the above constructor definition
    def __init__(self, x, y):
        print(f'Constructor second  with arguments {x}, {y}')


d1 = D(10, 20) # Constructor second   with arguments 10, 20
```

Constructor with Multiple Inheritance

We can't use super() to access all the superclasses in case of multiple inheritances. The better approach would be to call the constructor function of the superclasses using their class name.

```
class A1:
    def __init__(self, a1):
        print('A1 Constructor')
        self.var_a1 = a1


class B1:
    def __init__(self, b1):
        print('B1 Constructor')
        self.var_b1 = b1


class C1(A1, B1):
    def __init__(self, a1, b1, c1):
        print('C1 Constructor')
        A1.__init__(self, a1)
        B1.__init__(self, b1)
        self.var_c1 = c1


c_obj = C1(1, 2, 3)
print(f'c_obj var_a={c_obj.var_a1}, var_b={c_obj.var_b1}, var_c={c_obj.var_c1}')
```

Output:

```
C1 Constructor
A1 Constructor
B1 Constructor
c_obj var_a=1, var_b=2, var_c=3
```

Can Python __init__() function return something?

If we try to return a non-None value from the __init__() function, it will raise TypeError.

```
class Data:

   def __init__(self, i):
      self.id = i
      return True

d = Data(10)
```

Output:

```
1
TypeError: __init__() should return None, not 'bool'
```

If we change the return statement to return None then the code will work without any exception.

Constructor Chaining with Multilevel Inheritance

```
class A:

   def __init__(self, a):
      print('A Constructor')
      self.var_a = a


class B(A):

   def __init__(self, a, b):
      super().__init__(a)
      print('B Constructor')
      self.var_b = b


class C(B):

   def __init__(self, a, b, c):
      super().__init__(a, b)
      print('C Constructor')
      self.var_c = c


c_obj = C(1, 2, 3)
print(f'c_obj var_a={c_obj.var_a}, var_b={c_obj.var_b}, var_c={c_obj.var_c}')
```

Output:

A Constructor
B Constructor
C Constructor
c_obj var_a=1, var_b=2, var_c=3

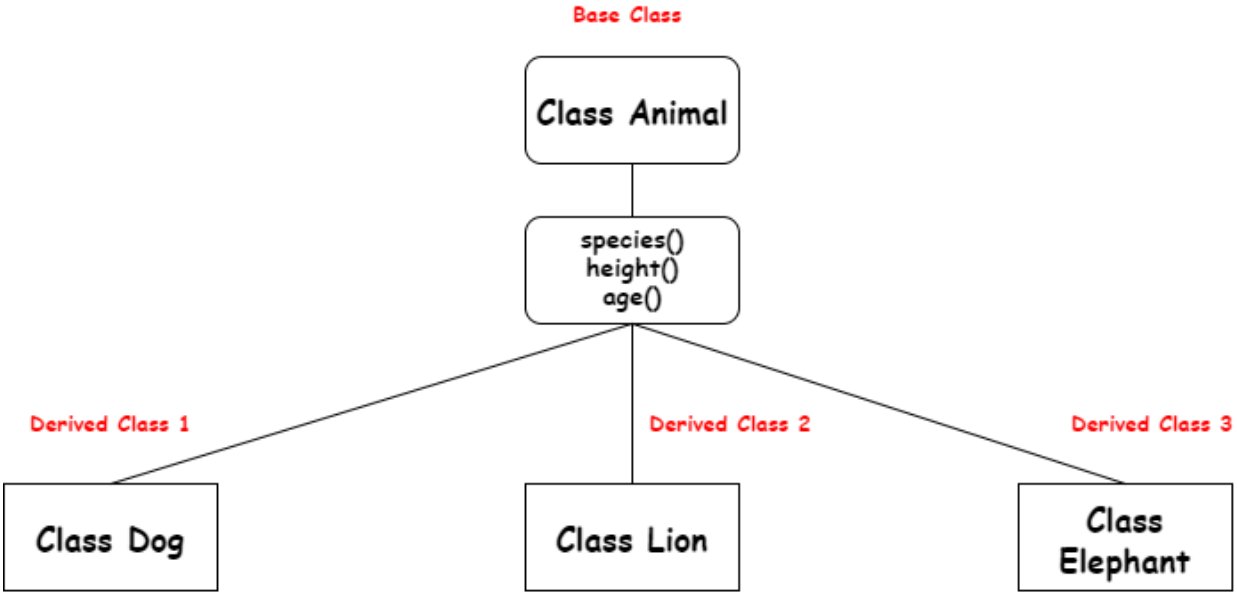
Inheritance

Inheritance is a way of creating a new class for using details of an existing class without modifying it. Inheritance enables us to define a class that takes all the functionality fro the parent class and allows us to add more. In object-oriented programming, inheritance enables new objects to take on the properties of existing objects.

A class that is used as the basis for inheritance is called a superclass or base class. A class that inherits from a superclass is called a subclass or derived class. The terms parent class and child class are also acceptable terms to use respectively. A child inherits visible properties and methods from its parent while adding additional properties and methods of its own. Subclasses and super classes can be understood in terms of the is a relationship. A subclass is a more specific instance of a superclass.

The basic syntax is as follows:

```
class BaseClass:
  "Body of base class"

class DerivedClass(BaseClass):
  "Body of derived class"
```




```
# Base class
class Dog:

    # Class attribute
    species = 'mammal'

    # Instance attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # instance method
    def description(self):
        return "{} is {} years old".format(self.name, self.age)

    # instance method
    def speak(self, sound):
        return "{} says {}".format(self.name, sound)


# Derived class (inherits from Dog class)
class Bulldog(Dog):
    def run(self, speed):
        return "{} runs {}".format(self.name, speed)


# Derived class inherits attributes and
# behavior from the parent class
```

```
Jim = Bulldog("Jim", 12)
print(Jim.description())

# Derived class has specific attributes
# and behavior as well
print(Jim.run("slowly"))
```

Output:

```
Jim is 12 years old
Jim runs slowly
```

Create a Base class

```
class Father:
   # The keyword 'self' is used to represent the instance of a class.
   # By using the "self" keyword we access the attributes and methods of the class in python.
   # The method "__init__"  is called as a constructor in object oriented terminology.
   # This method is called when an object is created from a class.
   # it allows the class to initialize the attributes of the class.
   def __init__(self, name, lastname):
      self.name = name
      self.lastname = lastname

   def printname(self):
      print(self.name, self.lastname)


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

x = Father("Haramohan", "Sahu")
x.printname()
```

Output: Haramohan Sahu

Create a Derived class

```
# The subclass __init__() function overrides the inheritance of the base class __init__() function.

class Son(Father):
   def __init__(self, name, lastname):
      Father.__init__(self, name, lastname)


x = Son("Haramohan", "Sahu")
x.printname()
```

OUTPUT:
Haramohan Sahu

Example 1: Use of Inheritance in Python

```
# parent class
class Bird:

    def __init__(self):
        print("Bird is ready")

    def whoisThis(self):
        print("Bird")

    def swim(self):
        print("Swim faster")

# child class
class Penguin(Bird):

    def __init__(self):
        # call super() function
        super().__init__()
        print("Penguin is ready")

    def whoisThis(self):
        print("Penguin")

    def run(self):
        print("Run faster")

peggy = Penguin()
peggy.whoisThis()
peggy.swim()
peggy.run()
```

Output

```
Bird is ready
Penguin is ready
Penguin
Swim faster
Run faster
```

Additionally, we use the super() function inside the __init__() method. This allows us to run the __init__() method of the parent class inside the child class.

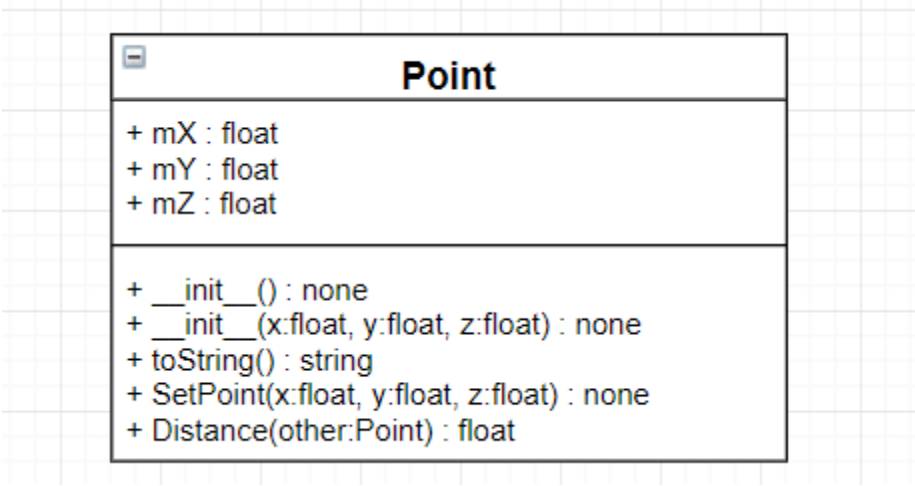
Use of super() function

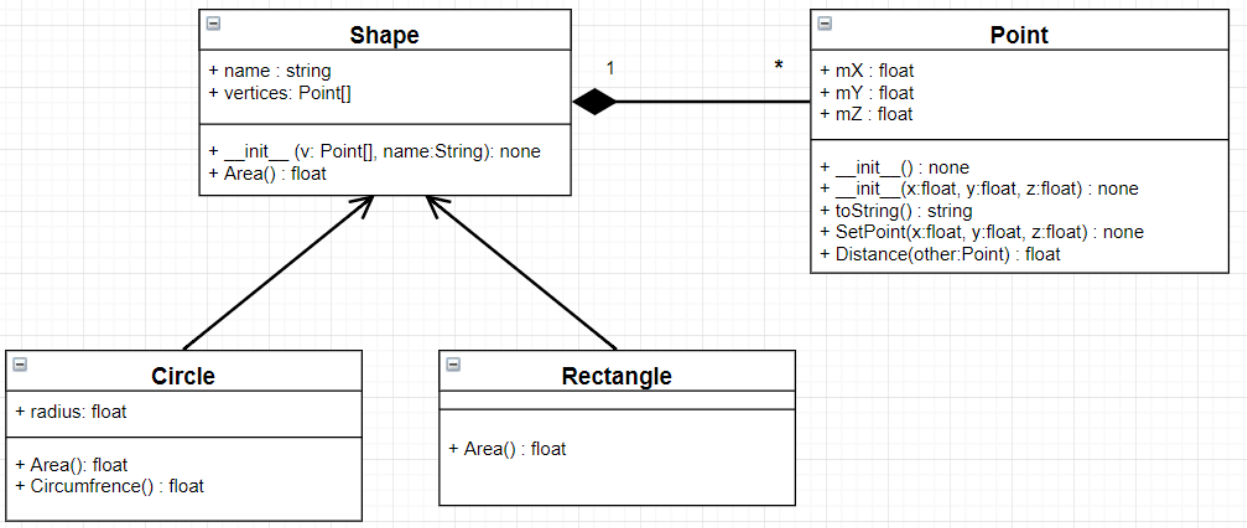
By using the super() function, you do not have to use the name of the parent element, it will automatically inherit the methods and properties from its parent.

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

  def printname(self):
    print(self.name, self.lastname)

class Son(Father):
  def __init__(self, name, lastname):
    super().__init__(name, lastname)

x = Student("Haramohan", "sahu")
x.printname()
```

Output:
Haramohan sahu

Some More example below with diagram





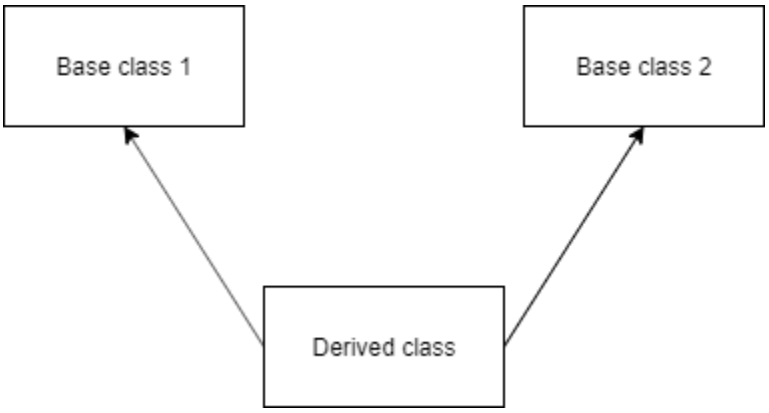

```
class Shape:
    ''' A general Shape class '''

    def __init__(self, p, n):
        '''Set the name and vertices of the shape'''
        self.name = n
        self.vertices = p


    def Area(self):
        '''Caculate the area of the shapes'''
        raise NotImplementedError("Please Implement the Area method!")
        #This containts a check to make sure subclasses implement this.
        return None

    def __str__(self):
        '''Definition for the print statement'''
        return "Shape: '{}' of type ({}) has {} points.".format(
            self.name,str(self.__class__.__name__),
            str(len(self.vertices)))



class Rectangle(Shape):
    '''Rectangle is a shape with 4 vertices'''

    def __init__(self, p, n):
```

```
        '''Rectangle vs: lower left, upper left, upper right, lower right'''
        if len(p) != 4:
            print('FATAL ERROR: A rectangle has 4 points.')
            return
        Shape.__init__(self,p,n)

    def Area(self):
        '''Area of a rectangle is length*width'''
        length = self.vertices[0].Distance(self.vertices[1])
        width = self.vertices[1].Distance(self.vertices[2])
        return length*width


def main():
    p1 = [Point(0.0,0.0,0.0), Point(0.0,3.0,0.0),
          Point(2.0,3.0,0.0), Point(2.0,0.0,0.0) ]
    r = Rectangle(p1,"Rectangle 1")
    print(r)
    print("Rectangle area is:", r.Area())


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

Python Multiple Inheritance

A class can be derived from more than one base classes in Python. This is called multiple inheritance. In multiple inheritance, the features of all the base classes are inherited into the derived class (see the figure below).

```
class Base1:
    pass

class Base2:
    pass

class MultiDerived(Base1, Base2):
    pass
```




```
# first parent class
class Manager(object):
    def __init__(self, name, idnumber):
        self.name = name
        self.idnumber = idnumber

# second parent class
class Employee(object):
    def __init__(self, salary, post):
        self.salary = salary
        self.post = post

# inheritance from both the parent classes
class Person(Manager, Employee):
    def __init__(self, name, idnumber, salary, post, points):
        self.points = points
        Manager.__init__(self, name, idnumber)
        Employee.__init__(self, salary, post)
        print(self.salary)

ins = Person('Rahul', 882016, 75000, 'Assistant Manager', 560)
```

Multiple resolution order

Method Resolution Order (MRO) is an approach that takes to resolve the variables or functions of a class.

In the multiple inheritance use case, the attribute is first looked up in the current class. If it fails, then the next place to search is in the parent class. If there are multiple parent classes, then the preference order is depth-first followed by a left-right path. MRO ensures that a class always precedes its parents and for multiple parents, keeps the order as the tuple of base classes and avoids ambiguity.

```
class Agile:
```

```
    def create(self):
        print(" Forming class Agile")


class Dev(Agile):
    def create(self):
        print(" Forming class Dev")


class QA(Agile):
    def create(self):
        print(" Forming class QA")


# Ordering of classes
class Sprint(Dev, QA):
    pass


sprint = Sprint()
sprint.create()
```

Output: Forming class Dev

Method Resolution Order(MRO)

Python provides a __mro__ attribute and the mro() method. With these, you can get the resolution order.

```
class Material:
    def create(self):
        print(" Creating class Appliance")


class Pencil:
    def create(self):
        print(" Creating class Pencil"
# Order of classes
 class Pen(Material, Pencil):
    def __init__(self):
        print("Constructing Pen")


appl = Pen()
```

```
# Display the lookup order
print(Pen.__mro__)
print(Pen.mro())
```
Output:

```
Constructing Pen
(<class '__main__.Pen'>, <class '__main__.Material'>, <class '__main__.Pencil'>, <class 'object'>)
[<class '__main__.Pen'>, <class '__main__.Material'>, <class '
```

Encapsulation

Using OOP in Python, we can restrict access to methods and variables. This prevents data from direct modification which is called encapsulation. Encapsulation means that the internal representation of an object is generally hidden from view outside of the object's definition.In Python, we denote private attributes using underscore as the prefix i.e single _ or double __.

Using Single Underscore

```
class Computer:

    def __init__(self):
        self.__maxprice = 900

    def sell(self):
        print("Selling Price: {}".format(self.__maxprice))

    def setMaxPrice(self, price):
        self.__maxprice = price

c = Computer()
c.sell()

# change the price
c.__maxprice = 1000
c.sell()

# using setter function
```

```
c.setMaxPrice(1000)
c.sell()
```

We tried to modify the price. However, we can't change it because Python treats the __maxprice as private attributes.

A common Python programming convention to identify a private variable is by prefixing it with an underscore. Now, this doesn't really make any difference on the compiler side of things.
The variable is still accessible as usual. But being a convention that programmers have picked up on, it tells other programmers that the variables or methods have to be used only within the scope of the class.

```
class Person:
   def __init__(self, name, age=0):
      self.name = name
      self._age = age

   def display(self):
      print(self.name)
      print(self._age)

person = Person('Dev', 30)
#accessing using class method
person.display()
#accessing directly from outside
print(person.name)
print(person._age)
```

Output

```
Dev
30
Dev
30
```

Here we can see that _age is accessible. As I Said earlier, it is just an convention.

Using Double Underscores

If you want to make class members i.e. methods and variables private, then you should prefix them with double underscores. But Python offers some sort of support to the private modifier. This mechanism is called Name mangling. With this, it is still possible to access the class members from outside it.

If you need to access the private member function

```
class SeeMee:
  def youcanseeme(self):
    return 'you can see me'

  def __youcannotseeme(self):
    return 'you cannot see me'

#Outside class
Check = SeeMee()
print(Check.youcanseeme())
print(Check._SeeMee__youcannotseeme())
#Changing the name causes it to access the function
```

what is name Mangling?

A double underscore prefix causes the Python interpreter to rewrite the attribute name in order to avoid naming conflicts in subclasses. This is also called name mangling—the interpreter changes the name of the variable in a way that makes it harder to create collisions when the class is extended later.

```
# Python program to demonstrate
# name mangling


class Student:
    def __init__(self, name):
        self.__name = name

s1 = Student("Santhosh")
print(dir(s1))
```

Output

['_Student__name', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__']

The above output shows dir() method returning all valid attributes with the name mangling process that is done to the class variable __name. The name changed from __name to _Student__name.

Abstract Classes and Methods in Python

To declare an Abstract class, we firstly need to import the abc module. Let us look at an example.

```
from abc import ABC
class abs_class(ABC):
    #abstract methods
```

As a property, abstract classes can have any number of abstract methods coexisting with any number of other methods. For example, we can see below.

```
from abc import ABC, abstractmethod
class abs_class(ABC):
   #normal method
   def method(self):
      #method definition
   @abstractmethod
   def Abs_method(self):
      #Abs_method definition
```

Here, method() is normal method whereas Abs_method() is an abstract method implementing @abstractmethod from the abc module.

Python Abstraction Example

```
from abc import ABC, abstractmethod
class Absclass(ABC):
   def print(self,x):
      print("Passed value: ", x)
   @abstractmethod
   def task(self):
      print("We are inside Absclass task")

class test_class(Absclass):
   def task(self):
      print("We are inside test_class task")

class example_class(Absclass):
   def print(self,x):
      print("example_class Passed value: ", x)
   def task(self):
      print("We are inside example_class task")

#object of test_class created
```

```
test_obj = test_class()
test_obj.task()
test_obj.print(100)

#object of example_class created
example_obj = example_class()
example_obj.task()
example_obj.print(200)


print("test_obj is instance of Absclass? ", isinstance(test_obj, Absclass))
print("example_obj is instance of Absclass? ", isinstance(example_obj, Absclass))
```

OUTPUT:
We are inside test_class task
Passed value: 100
We are inside example_class task
example_class Passed value: 200
test_obj is instance of Absclass? True
example_obj is instance of Absclass? True

Note: We cannot create instances of an abstract class. It raises an Error.

Polymorphism in Python

This phenomenon refers to the ability to be able to display in multiple forms.

Suppose, we need to color a shape. There are multiple shape options (rectangle, square, circle). However, we could use the same method to color any shape. This concept is called Polymorphism.

```
class Rectangle:
   def draw(self):
      print("Drawing a rectangle.")

 class Triangle:
    def draw(self):
      print("Drawing a Triangle.")


# common interface
def draw_shape(Shape):
   Shape.draw()

#instantiate objects
```

```
A = Rectangle()
B = Triangle()

# passing the object
draw_shape(A)
draw_shape(B)
```

Output:

```
Drawing a Rectangle.
Drawing a Triangle.
```

Implementing Polymorphism in Python with Class

Python can use different types of classes, in the same way, using Polymorphism.
To serve this purpose, one can create a loop that iterates through a tuple of objects. Post which, one can call the methods without having a look at the type of class to which the object belongs to.

Example: Polymorphism with Classes and Objects

```
class Rabbit():
   def age(self):
      print("This function determines the age of Rabbit.")

   def color(self):
      print("This function determines the color of Rabbit.")

class Horse():
   def age(self):
      print("This function determines the age of Horse.")

   def color(self):
      print("This function determines the color of Horse.")

obj1 = Rabbit()
obj2 = Horse()
for type in (obj1, obj2): # creating a loop to iterate through the obj1, obj2
   type.age()
   type.color()
```

Output:

```
This function determines the age of Rabbit.
This function determines the color of Rabbit.
```

This function determines the age of Horse.
This function determines the color of Horse.

Implementing Polymorphism in Python with Inheritance

We will be defining functions in the derived class that has the same name as the functions in the base class. Here, we re-implement the functions in the derived class. The phenomenon of re-implementing a function in the derived class is known as Method Overriding.

Example: Polymorphism with Inheritance

```
class Animal:
  def type(self):
    print("Various types of animals")

  def age(self):
    print("Age of the animal.")

class Rabbit(Animal):
  def age(self):
    print("Age of rabbit.")

class Horse(Animal):
  def age(self):
    print("Age of horse.")

obj_animal = Animal()
obj_rabbit = Rabbit()
obj_horse = Horse()

obj_animal.type()
obj_animal.age()

obj_rabbit.type()
obj_rabbit.age()

obj_horse.type()
obj_horse.age()
```

Output:

```
Various types of animals
Age of the animal.
Various types of animals
Age of rabbit.
Various types of animals
```

Age of horse.

Compile-Time Polymorphism or Method Overloading?

Unlike many other popular object-oriented programming languages such as Java, Python doesn't support compile-time polymorphism or method overloading. If a class or Python script has multiple methods with the same name, the method defined in the last will override the earlier one.

Python doesn't use function arguments for method signature, that's why method overloading is not supported in Python.

Working of Python NONE object

Python NONE is an object in the world of Python

The NONE object is of data type 'NoneType' and hence, it can not be considered as a value of certain primitive data types or boolean values.

example:

```
var = None
print("Value of var: ",var) #Value of var:  None
var = None
print("Boolean Check on NONE keyword:\n")
if var:
  print("TRUE")
else:
  print("FALSE")  # this will be executed


var_set = {10,None,30,40,50}
for x in var_set:
    print(x)

OUTPUT:
40
50
10
30
None

var_lst = [10,None,30,40,50]
for x in var_lst:
    print(str(x))
Output:
```

10
None
30
40
50

Overloading the + Operator

To overload the + operator, we will need to implement __add__() function in the class. With great power comes great responsibility.
We can do whatever we like, inside this function. But it is more sensible to return a Point object of the coordinate sum.

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __str__(self):
        return "({0},{1})".format(self.x, self.y)

    def __add__(self, other):
        x = self.x + other.x
        y = self.y + other.y
        return Point(x, y)


p1 = Point(1, 2)
p2 = Point(2, 3)

print(p1+p2)
```

Output

(3,5)

Similarly, we can overload other operators as well. The special function that we need to implement is tabulated below.

Operator	Expression	Internally
Addition	p1 + p2	p1.__add__(p2)
Subtraction	p1 - p2	p1.__sub__(p2)
Multiplication	p1 * p2	p1.__mul__(p2)
Power	p1 ** p2	p1.__pow__(p2)
Division	p1 / p2	p1.__truediv__(p2)
Floor Division	p1 // p2	p1.__floordiv__(p2)
Remainder (modulo)	p1 % p2	p1.__mod__(p2)
Bitwise Left Shift	p1 << p2	p1.__lshift__(p2)
Bitwise Right Shift	p1 >> p2	p1.__rshift__(p2)
Bitwise AND	p1 & p2	p1.__and__(p2)
Bitwise OR	p1 | p2	p1.__or__(p2)
Bitwise XOR	p1 ^ p2	p1.__xor__(p2)
Bitwise NOT	~p1	p1.__invert__()

Similarly, the special functions that we need to implement, to overload other comparison operators are tabulated below.

Operator	Expression	Internally
Less than	p1 < p2	p1.__lt__(p2)
Less than or equal to	p1 <= p2	p1.__le__(p2)
Equal to	p1 == p2	p1.__eq__(p2)
Not equal to	p1 != p2	p1.__ne__(p2)
Greater than	p1 > p2	p1.__gt__(p2)
Greater than or equal to	p1 >= p2	p1.__ge__(p2)

Python Iterators

Iterators are objects that can be iterated upon.

Iterators are everywhere in Python. They are elegantly implemented within for loops, comprehensions, generators etc. but are hidden in plain sight.

Iterator in Python is simply an object that can be iterated upon. An object which will return data, one element at a time.

Technically speaking, a Python iterator object must implement two special methods, __iter__() and __next__(), collectively called the iterator protocol.

An object is called iterable if we can get an iterator from it. Most built-in containers in Python like: list, tuple, string etc. are iterables.

The iter() function (which in turn calls the __iter__() method) returns an iterator from them.

We use the next() function to manually iterate through all the items of an iterator. When we reach the end and there is no more data to be returned, it will raise the StopIteration Exception.
Following is an example.

```
# define a list
my_list = [4, 7, 0, 3]

# get an iterator using iter()
my_iter = iter(my_list)

# iterate through it using next()

# Output: 4
print(next(my_iter))

# Output: 7
print(next(my_iter))

# next(obj) is same as obj.__next__()

# Output: 0
print(my_iter.__next__())

# Output: 3
print(my_iter.__next__())

# This will raise error, no items left
next(my_iter)
```

Iterators in Python

Iterators are everywhere in Python. They are elegantly implemented within for loops, comprehensions, generators etc. but are hidden in plain sight.

Iterator in Python is simply an object that can be iterated upon. An object which will return data, one element at a time.

Technically speaking, a Python iterator object must implement two special methods, __iter__() and __next__(), collectively called the iterator protocol.

An object is called iterable if we can get an iterator from it. Most built-in containers in Python like: list, tuple, string etc. are iterables.

The iter() function (which in turn calls the __iter__() method) returns an iterator from them.

```
# define a list
my_list = [4, 7, 0, 3]

# get an iterator using iter()
my_iter = iter(my_list)

# iterate through it using next()

# Output: 4
print(next(my_iter))

# Output: 7
print(next(my_iter))

# next(obj) is same as obj.__next__()

# Output: 0
print(my_iter.__next__())

# Output: 3
print(my_iter.__next__())

# This will raise error, no items left
next(my_iter)
```

Run Code

Output

```
Traceback (most recent call last):
  File "<string>", line 24, in <module>
    next(my_iter)
StopIteration
```

A more elegant way of automatically iterating is by using the for loop. Using this, we can iterate over any object that can return an iterator,

```
>>> for element in my_list:
```

```
...     print(element)
...
4
7
0
3
```

Building Custom Iterators

Building an iterator from scratch is easy in Python. We just have to implement the __iter__() and the __next__() methods.

The __iter__() method returns the iterator object itself. If required, some initialization can be performed.

The __next__() method must return the next item in the sequence. On reaching the end, and in subsequent calls, it must raise StopIteration.

example

```
class PowTwo:
   """Class to implement an iterator
   of powers of two"""

   def __init__(self, max=0):
      self.max = max

   def __iter__(self):
      self.n = 0
      return self

   def __next__(self):
      if self.n <= self.max:
         result = 2 ** self.n
         self.n += 1
         return result
      else:
         raise StopIteration


# create an object
numbers = PowTwo(3)

# create an iterable from the object
i = iter(numbers)
```

```
# Using next to get to the next iterator element
# Using next to get to the next iterator element
print(next(i))
print(next(i))
print(next(i))
print(next(i))
print(next(i))
```

```
1
2
4
8
Traceback (most recent call last):
  File "/home/bsoyuj/Desktop/Untitled-1.py", line 32, in <module>
    print(next(i))
  File "<string>", line 18, in __next__
    raise StopIteration
StopIteration
```

Python generator:

Generator functions allow you to declare a function that behaves like an iterator.
They allow programmers to make an iterator in a fast, easy, and clean way.
Generators are used to create iterators, but with a different approach. Generators are simple functions which return an iterable set of items, one at a time, in a special way. ... The generator function can generate as many values (possibly infinite) as it wants, yielding each one in its turn.

There is a lot of work in building an iterator in Python. We have to implement a class with __iter__() and __next__() method, keep track of internal states, and raise StopIteration when there are no values to be returned. This is both lengthy and counterintuitive. Generator comes to the rescue in such situations. Python generators are a simple way of creating iterators. All the work we mentioned above are automatically handled by generators in Python.

Simply speaking, a generator is a function that returns an object (iterator) which we can iterate over (one value at a time).

```
# A simple generator function
def my_gen():
    n = 1
    print('This is printed first')
    # Generator function contains yield statements
    yield n
```

```
    n += 1
    print('This is printed second')
    yield n

    n += 1
    print('This is printed at last')
    yield n


# Using for loop
for item in my_gen():
    print(item)
```

```
This is printed first
1
This is printed second
2
This is printed at last
3
```

Let's take an example of a generator that reverses a string.

```
def rev_str(my_str):
    length = len(my_str)
    for i in range(length - 1, -1, -1):
        yield my_str[i]


# For loop to reverse the string
for char in rev_str("hello"):
    print(char)
```

Output

```
o
l
l
e
h
```

Differences between Generator function and Normal function

Here is how a generator function differs from a normal function.

Generator function contains one or more yield statements.
When called, it returns an object (iterator) but does not start execution immediately.
Methods like __iter__() and __next__() are implemented automatically. So we can iterate through the items using next().
Once the function yields, the function is paused and the control is transferred to the caller.
Local variables and their states are remembered between successive calls.
Finally, when the function terminates, StopIteration is raised automatically on further calls.

More Examples on Generators:

Here is an example to illustrate all of the points stated above. We have a generator function named my_gen() with several yield statements.

```
# A simple generator function
def my_gen():
    n = 1
    print('This is printed first')
    # Generator function contains yield statements
    yield n

    n += 1
    print('This is printed second')
    yield n

    n += 1
    print('This is printed at last')
    yield n
```

An interactive run in the interpreter is given below. Run these in the Python shell to see the output.

```
>>> # It returns an object but does not start execution immediately.
>>> a = my_gen()

>>> # We can iterate through the items using next().
>>> next(a)
This is printed first
1
>>> # Once the function yields, the function is paused and the control is transferred to the caller.

>>> # Local variables and theirs states are remembered between successive calls.
>>> next(a)
```

```
This is printed second
2

>>> next(a)
This is printed at last
3

>>> # Finally, when the function terminates, StopIteration is raised automatically on further calls.
>>> next(a)
Traceback (most recent call last):
...
StopIteration
>>> next(a)
Traceback (most recent call last):
...
StopIteration
```

One interesting thing to note in the above example is that the value of variable n is remembered between each call.

Unlike normal functions, the local variables are not destroyed when the function yields. Furthermore, the generator object can be iterated only once.

To restart the process we need to create another generator object using something like a = my_gen().

One final thing to note is that we can use generators with for loops directly.

This is because a for loop takes an iterator and iterates over it using next() function. It automatically ends when StopIteration is raised. Check here to know how a for loop is actually implemented in Python.

 # A simple generator function

```
def my_gen():
    n = 1
    print('This is printed first')
    # Generator function contains yield statements
    yield n

    n += 1
    print('This is printed second')
    yield n

    n += 1
    print('This is printed at last')
```

```
    yield n


# Using for loop
for item in my_gen():
    print(item)
```

OUTPUT

```
This is printed first
1
This is printed second
2
This is printed at last
3
```

Use of Python Generators

There are several reasons that make generators a powerful implementation.

1. Easy to Implement

Generators can be implemented in a clear and concise way as compared to their iterator class counterpart. Following is an example to implement a sequence of power of 2 using an iterator class.

```
class PowTwo:
    def __init__(self, max=0):
        self.n = 0
        self.max = max

    def __iter__(self):
        return self

    def __next__(self):
        if self.n > self.max:
            raise StopIteration

        result = 2 ** self.n
        self.n += 1
        return result
```

The above program was lengthy and confusing. Now, let's do the same using a generator function.

```
def PowTwoGen(max=0):
    n = 0
```

```
    while n < max:
        yield 2 ** n
        n += 1
```

Since generators keep track of details automatically, the implementation was concise and much cleaner.

2. Memory Efficient

A normal function to return a sequence will create the entire sequence in memory before returning the result. This is an overkill, if the number of items in the sequence is very large.

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.

The following generator function can generate all the even numbers (at least in theory).

```
def all_even():
    n = 0
    while True:
        yield n
        n += 2
```

Python Closures

A Closure is a function object that remembers values in enclosing scopes even if they are not present in memory. A Closure is a function object that remembers values in enclosing scopes even if they are not present in memory.

It is a record that stores a function together with an environment: a mapping associating each free variable of the function (variables that are used locally, but defined in an enclosing scope) with the value or reference to which the name was bound when the closure was created.
A closure—unlike a plain function—allows the function to access those captured variables through the closure's copies of their values or references, even when the function is invoked outside their scope.
Three characteristics of a Python closure are:

- it is a nested function
- it has access to a free variable in outer scope
- it is returned from the enclosing function

When and why to use Closures:

As closures are used as callback functions, they provide some sort of data hiding. This helps us to reduce the use of global variables.

When we have few functions in our code, closures prove to be efficient way. But if we need to have many functions, then go for class (OOP).
When there are few methods (one method in most cases) to be implemented in a class, closures can provide an alternate and more elegant solution. But when the number of attributes and methods get larger, it's better to implement a class.

Python Decorators make an extensive use of closures as well.

Here is a simple example where a closure might be more preferable than defining a class and making objects. But the preference is all yours.

```
def make_multiplier_of(n):
    def multiplier(x):
        return x * n
    return multiplier


# Multiplier of 3
times3 = make_multiplier_of(3)

# Multiplier of 5
times5 = make_multiplier_of(5)

# Output: 27
print(times3(9))

# Output: 15
print(times5(3))

# Output: 30
print(times5(times3(2)))
```

some more

```
# Python program to illustrate
# closures
import logging
logging.basicConfig(filename='example.log', level=logging.INFO)


def logger(func):
    def log_func(*args):
        logging.info(
            'Running "{}" with arguments {}'.format(func.__name__, args))
        print(func(*args))
```

```
    # Necessary for closure to work (returning WITHOUT parenthesis)
    return log_func

def add(x, y):
    return x+y

def sub(x, y):
    return x-y

add_logger = logger(add)
sub_logger = logger(sub)

add_logger(3, 3)
add_logger(4, 5)

sub_logger(10, 5)
sub_logger(20, 10)
```

OUTPUT:

```
6
9
5
10
```

All function objects have a __closure__ attribute that returns a tuple of cell objects if it is a closure function. Referring to the example above, we know times3 and times5 are closure functions.

```
>>> make_multiplier_of.__closure__
>>> times3.__closure__
(<cell at 0x0000000002D155B8: int object at 0x000000001E39B6E0>,)
```

The cell object has the attribute cell_contents which stores the closed value.

```
>>> times3.__closure__[0].cell_contents
3
>>> times5.__closure__[0].cell_contents
5
```

Python closure with nonlocal keyword

The nonlocal keyword allows us to modify a variable with immutable type in the outer function scope.

```
ef make_counter():

    count = 0
```

```
    def inner():

        nonlocal count
        count += 1
        return count

    return inner


counter = make_counter()

c = counter()
print(c)

c = counter()
print(c)

c = counter()
print(c)
1
2
3
```

Python Decorators

A decorator in Python is any callable Python object that is used to modify a function or a class. A reference to a function "func" or a class "C" is passed to a decorator and the decorator returns a modified function or class.

Python's decorators allow you to extend and modify the behavior of a callable (functions, methods, and classes) without permanently modifying the callable itself. It add functionality to an existing code. Functions and methods are called callable as they can be called. In fact, any object which implements the special __call__() method is termed callable. So, in the most basic sense, a decorator is a callable that returns a callable.
Basically, a decorator takes in a function, adds some functionality and returns it.

```
def make_pretty(func):
    def inner():
        print("I got decorated")
        func()
    return inner
```

```
def ordinary():
   print("I am ordinary")



pretty =make_pretty(ordinary);
pretty ();
```

OUTPUT:
I got decorated
I am ordinary

The function ordinary() got decorated and the returned function was given the name pretty .

Generally, we decorate a function and reassign it as,

ordinary = make_pretty(ordinary).
This is a common construct and for this reason, Python has a syntax to simplify this.

We can use the @ symbol along with the name of the decorator function and place it above the definition of the function to be decorated. For example,

```
@make_pretty
def ordinary():
   print("I am ordinary")
```
is equivalent to

```
def ordinary():
   print("I am ordinary")
ordinary = make_pretty(ordinary)
```

Now let's make a decorator to check for this case that will cause the error.

```
def smart_divide(func):
   def inner(a, b):
      print("I am going to divide", a, "and", b)
      if b == 0:
         print("Whoops! cannot divide")
         return

      return func(a, b)
   return inner
```

```
@smart_divide
def divide(a, b):
    print(a/b)

divide(2,5)
```

some more example of Decorator with any number of parameters:

```
def star(func):
    def inner(*args, **kwargs):
        print("*" * 30)
        func(*args, **kwargs)
        print("*" * 30)
    return inner


def percent(func):
    def inner(*args, **kwargs):
        print("%" * 30)
        func(*args, **kwargs)
        print("%" * 30)
    return inner


@star
@percent
def printer(msg):
    print(msg)


printer("Hello")
```

Output

```
******************************
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Hello
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
******************************
```

The above syntax of,

```
@star
@percent
def printer(msg):
   print(msg)
```
is equivalent to

```
def printer(msg):
   print(msg)
printer = star(percent(printer))
```

The following example will illustrate the significance of the call decorator method. In this example, we would be building a list of the doubles of the first “n” numbers, using a helper function.
The code is as follows:

```
# Helper function to build a
# list of numbers
def list_of_numbers(n):
   element = []
   for i in range(n):
      element.append(i * 2)
   return element

list_of_numbers = list_of_numbers(6)

# Output command
print(len(list_of_numbers),
     list_of_numbers[2])
```

Output

6, 4

The above code could also be written using the call() decorator:

```
# Defining the decorator function
def call(*argv, **kwargs):
   def call_fn(function):
      return function(*argv, **kwargs)
   return call_fn

# Using the decorator function
@call(6)
def list_of_numbers(n):
```

```
    element = []
    for i in range(n):
        element.append(i * 2)
    return element

# Output command
print(len(list_of_numbers),
      list_of_numbers[2])
```

Output

6, 4

Python @property decorator

python programming provides us with a built-in @property decorator which makes usage of getter and setters much easier in Object-Oriented Programming.
Before going into details on what @property decorator is, let us first build an intuition on why it would be needed in the first place.

with getter & setter method:

```
# Making Getters and Setter methods
class Celsius:
    def __init__(self, temperature=0):
        self.set_temperature(temperature)

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

    # getter method
    def get_temperature(self):
        return self.temperature

    # setter method
    def set_temperature(self, value):
        if value < -273.15:
            raise ValueError("Temperature below -273.15 is not possible.")
        self.temperature = value


# Create a new object, set_temperature() internally called by __init__
human = Celsius(37)
```

```
# Get the temperature attribute via a getter
print(human.get_temperature())

# Get the to_fahrenheit method, get_temperature() called by the method itself
print(human.to_fahrenheit())
```

```
37
98.60000000000001
```

The above program with The property Class

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

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

    # getter
    def get_temperature(self):
        print("Getting value...")
        return self._temperature

    # setter
    def set_temperature(self, value):
        print("Setting value...")
        if value < -273.15:
            raise ValueError("Temperature below -273.15 is not possible")
        self._temperature = value

    # creating a property object
    temperature = property(get_temperature, set_temperature)



human = Celsius(37)

print(human.temperature)

print(human.to_fahrenheit())

human.temperature = -300
```

```
Setting value...
Getting value...
37
Getting value...
98.60000000000001
```

temperature = property(get_temperature,set_temperature)
can be broken down as:

```
# make empty property
temperature = property()
# assign fget
temperature = temperature.getter(get_temperature)
# assign fset
temperature = temperature.setter(set_temperature)
```

These two pieces of codes are equivalent.

same programm with @poperty

For this, we reuse the temperature name while defining our getter and setter functions. Let's look at how to implement this as a decorator:

```
# Using @property decorator
class Celsius:
    def __init__(self, temperature=0):
        self.temperature = temperature

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

    @property
    def temperature(self):
        print("Getting value...")
        return self._temperature

    @temperature.setter
    def temperature(self, value):
        print("Setting value...")
        if value < -273.15:
```

```
            raise ValueError("Temperature below -273 is not possible")
        self._temperature = value


# create an object
human = Celsius(37)

print(human.temperature)

print(human.to_fahrenheit())

coldest_thing = Celsius(-300)
```

Note:
Whenever we assign or retrieve any object attribute like temperature as shown above, Python searches it in the object's built-in __dict__ dictionary attribute.

@classmethod decorator

```
class person:
    totalObjects=0
    def __init__(self):
        person.totalObjects=person.totalObjects+1
        self.n=89;

    @classmethod
    def showcount(cls):
        print("Total objects: ",cls.totalObjects)
        #print(cls.n)  #it can not be accessed by cls as it is instance variable


p= person();
p.showcount()
```

@staticmethod decorator

The @staticmethod is a built-in decorator that defines a static method in the class in Python.
A static method doesn't receive any reference argument whether it is called by an instance of a class or by the class itself.
The following notation is used to declare a static method in a class:
Example: Define Static Method Copy

```
class person:
   @staticmethod
   def greet():
      print("Hello!")


>>> person.greet()
Hello!
>>> p1=person()
>>> p1.greet()
Hello!
```

Even though both person.greet() and p.greet() are valid calls, the static method receives reference of neither. Hence it doesn't have any arguments - neither self nor cls.

Magic Methods In python:

Use the dir() function to see the number of magic methods inherited by a class.
Dir(int) will tell all the magic methods related to integer class. For more info, please follow below link
https://rszalski.github.io/magicmethods/

Enumerate() in Python

A lot of times when dealing with iterators, we also get a need to keep a count of iterations.
Python eases the programmers' task by providing a built-in function enumerate() for this task.
Enumerate() method adds a counter to an iterable and returns it in a form of enumerate object.
This enumerate object can then be used directly in for loops or be converted into a list of tuples using list() method.

```
# Python program to illustrate
# enumerate function in loops
l1 = ["eat","sleep","repeat"]

# printing the tuples in object directly
for ele in enumerate(l1):
   print ele
print
# changing index and printing separately
for count, ele in enumerate(l1,100):
   print count, ele
```

Output:

```
(0, 'eat')
(1, 'sleep')
(2, 'repeat')
```

100 eat
101 sleep
102 repeat

Threads in Python:

Running several threads is similar to running several different programs concurrently, but with the following benefits −

Multiple threads within a process share the same data space with the main thread and can therefore share information or communicate with each other more easily than if they were separate processes.

Threads are sometimes called light-weight processes and they do not require much memory overhead; they are cheaper than processes.
A thread has a beginning, an execution sequence, and a conclusion. It has an instruction pointer that keeps track of where within its context
is it currently running.
It can be pre-empted (interrupted).
It can temporarily be put on hold (also known as sleeping) while other threads are running - this is called yielding.
There are two different kind of threads −

- kernel thread
- user thread

There are two modules which support the usage of threads in Python3 −

_thread → old one
threading
The thread module has been "deprecated" for quite a long time. Users are encouraged to use the threading module instead. Hence, in Python 3,
the module "thread" is not available anymore. However, it has been renamed to "_thread" for backwards compatibilities in Python3.

Starting a new Thread using threading.thread.

Thread module emulating a subset of Java's threading model.

threading.Thread(group=None, target=None, name=None, args=(), kwargs={}, *, daemon=None)

- leave **group** as None.
- **target** is the callable object to be invoked by the run() method of Thread.
- **name** is the Thread name that you can provide and refer to later in the program.
- **args** is the argument tuple for the target invocation.
- **kwargs** is a dictionary of keyword arguments for the target invocation.

- **daemon** is not None, will be set to be daemonic.

Start a Thread

Once you have created a thread using Thread() class, you can start it using Thead.start() method.

t1 = threading.Thread()

t1.start()

Wait until the thread is finished

threading.Thread().join()

The simplest way to use a Thread is to instantiate it with a target function and call start() to let it begin working.

```
import threading

def worker():
    """thread worker function"""
    print 'Worker'
    return

threads = []
for i in range(5):
    t = threading.Thread(target=worker)
    threads.append(t)
    t.start()
```

The output is five lines with `"Worker"` on each:

```
$ python threading_simple.py

Worker
Worker
Worker
Worker
Worker
```

It useful to be able to spawn a thread and pass it arguments to tell it what work to do. This example passes a number, which the thread then prints.

```
import threading
def worker(num):
    """thread worker function"""
    print 'Worker: %s' % num
    return

threads = []
for i in range(5):
```

```
    t = threading.Thread(target=worker, args=(i,))
    threads.append(t)
    t.start()
```

OUTPUT:
python -u threading_simpleargs.py

```
Worker: 0
Worker: 1
Worker: 2
Worker: 3
Worker: 4
```

Determining the Current Thread

```
import threading
import time

def worker():
    print threading.currentThread().getName(), 'Starting'
    time.sleep(2)
    print threading.currentThread().getName(), 'Exiting'

def my_service():
    print threading.currentThread().getName(), 'Starting'
    time.sleep(3)
    print threading.currentThread().getName(), 'Exiting'

t = threading.Thread(name='my_service', target=my_service)
w = threading.Thread(name='worker', target=worker)
w2 = threading.Thread(target=worker) # use default name

w.start()
w2.start()
t.start()
```

OUTPUT:

```
worker Thread-1 Starting
my_service Starting
Starting
Thread-1worker Exiting
 Exiting
my_service Exiting
```

The logging module supports embedding the thread name in every log message using the formatter code %(threadName)s. Including thread names in log messages makes it easier to trace those messages back to their source.

```
import logging
import threading
import time

logging.basicConfig(level=logging.DEBUG,
                    format='[%(levelname)s] (%(threadName)-10s) %(message)s',
                    )

def worker():
    logging.debug('Starting')
    time.sleep(2)
    logging.debug('Exiting')

def my_service():
    logging.debug('Starting')
    time.sleep(3)
    logging.debug('Exiting')

t = threading.Thread(name='my_service', target=my_service)
w = threading.Thread(name='worker', target=worker)
w2 = threading.Thread(target=worker) # use default name

w.start()
w2.start()
t.start()
```

OUTPUT:

```
[DEBUG] (worker    ) Starting
[DEBUG] (Thread-1  ) Starting
[DEBUG] (my_service) Starting
[DEBUG] (worker    ) Exiting
[DEBUG] (Thread-1  ) Exiting
[DEBUG] (my_service) Exiting
```

Daemon vs. Non-Daemon Threads

To wait until a daemon thread has completed its work, use the join() method.

```
import threading
import time
import logging
```

```
logging.basicConfig(level=logging.DEBUG,
                    format='(%(threadName)-10s) %(message)s',
                    )

def daemon():
   logging.debug('Starting')
   time.sleep(2)
   logging.debug('Exiting')

d = threading.Thread(name='daemon', target=daemon)
d.setDaemon(True)

def non_daemon():
   logging.debug('Starting')
   logging.debug('Exiting')

t = threading.Thread(name='non-daemon', target=non_daemon)
d.start()
t.start()

d.join()
t.join()
```

Waiting for the daemon thread to exit using join() means it has a chance to produce its "Exiting" message.

OUTPUT

```
$ python threading_daemon_join.py

(daemon    ) Starting
(non-daemon) Starting
(non-daemon) Exiting
(daemon    ) Exiting
```

Enumerating All Threads

It is not necessary to retain an explicit handle to all of the daemon threads in order to ensure they have completed before exiting the main process. enumerate() returns a list of active Thread instances. The list includes the current thread, and since joining the current thread is not allowed (it introduces a deadlock situation), it must be skipped.

```
import random
import threading
import time
```

```
import logging

logging.basicConfig(level=logging.DEBUG,
                    format='(%(threadName)-10s) %(message)s',
                    )

def worker():
    """thread worker function"""
    t = threading.currentThread()
    pause = random.randint(1,5)
    logging.debug('sleeping %s', pause)
    time.sleep(pause)
    logging.debug('ending')
    return

for i in range(3):
    t = threading.Thread(target=worker)
    t.setDaemon(True)
    t.start()

main_thread = threading.currentThread()
for t in threading.enumerate():
    if t is main_thread:
        continue
    logging.debug('joining %s', t.getName())
    t.join()
```

Since the worker is sleeping for a random amount of time, the output from this program may vary. It should look something like this:

OUTPUT

```
$ python threading_enumerate.py

(Thread-1  ) sleeping 3
(Thread-2  ) sleeping 2
(Thread-3  ) sleeping 5
(MainThread) joining Thread-1
(Thread-2  ) ending
(Thread-1  ) ending
(MainThread) joining Thread-3
(Thread-3  ) ending
(MainThread) joining Thread-2
```

Subclassing Thread in Python:

At start-up, a Thread does some basic initialization and then calls its run() method, which calls the target function passed to the constructor. To create a subclass of Thread, override run() to do whatever is necessary.To implement a new thread using the threading module, you have to do the following −

Define a new subclass of the Thread class.

Override the __init__(self [,args]) method to add additional arguments.

Then, override the run(self [,args]) method to implement what the thread should do when started.

Once you have created the new Thread subclass, you can create an instance of it and then start a new thread by invoking the start(), which in turn calls the run() method.

```
import threading
import logging

logging.basicConfig(level=logging.DEBUG,
                    format='(%(threadName)-10s) %(message)s',
                    )
class MyThread(threading.Thread):
   def run(self):
      logging.debug('running')
      return

for i in range(5):
   t = MyThread()
   t.start()
```

The return value of run() is ignored.

OUTPUT:

```
$ python threading_subclass.py

(Thread-1  ) running
(Thread-2  ) running
(Thread-3  ) running
(Thread-4  ) running
(Thread-5  ) running
```

```
import threading
import logging

logging.basicConfig(level=logging.DEBUG,
                    format='(%(threadName)-10s) %(message)s',
                    )

class MyThreadWithArgs(threading.Thread):

    def __init__(self, group=None, target=None, name=None,
                 args=(), kwargs=None):
        threading.Thread.__init__(self, group=group, target=target, name=name,
                                  )
        self.args = args
        self.kwargs = kwargs
        return

    def run(self):
        logging.debug('running with %s and %s', self.args, self.kwargs)
        return

for i in range(5):
    t = MyThreadWithArgs(args=(i,), kwargs={'a':'A', 'b':'B'})
    t.start()
```

here we can remove the default values in init too.

OUTPUT:

```
(Thread-1  ) running with (0,) and {'a': 'A', 'b': 'B'}
>>> (Thread-2  ) running with (1,) and {'a': 'A', 'b': 'B'}
(Thread-3  ) running with (2,) and {'a': 'A', 'b': 'B'}
(Thread-4  ) running with (3,) and {'a': 'A', 'b': 'B'}
(Thread-5  ) running with (4,) and {'a': 'A', 'b': 'B'}
```

Timer Threads

One example of a reason to subclass Thread is provided by Timer, also included in threading. A Timer starts its work after a delay, and can be canceled at any point within that delay time period.

```
import threading
import time
import logging
```

```
logging.basicConfig(level=logging.DEBUG,
                format='(%(threadName)-10s) %(message)s',
                )

def delayed():
   logging.debug('worker running')
   return

t1 = threading.Timer(3, delayed)
t1.setName('t1')
t2 = threading.Timer(3, delayed)
t2.setName('t2')

logging.debug('starting timers')
t1.start()
t2.start()

logging.debug('waiting before canceling %s', t2.getName())
time.sleep(2)
logging.debug('canceling %s', t2.getName())
t2.cancel()
logging.debug('done')
```

OUTPUT:

```
(MainThread) starting timers
(MainThread) waiting before canceling t2
(MainThread) canceling t2
(MainThread) done
>>> (t1        ) worker running
```

Example – Python Mutlithreading with Two Threads

```
import threading
 def print_one():
            for i in range(10):
                        print(1)

def print_two():
            for i in range(10):
                        print(2)

if __name__ == "__main__":
            # create threads
```

```
    t1 = threading.Thread(target=print_one)
    t2 = threading.Thread(target=print_two)

    # start thread 1
    t1.start()
    # start thread 2
    t2.start()

    # wait until thread 1 is completely executed
    t1.join()
    # wait until thread 2 is completely executed
    t2.join()
    # both threads completely executed

    print("Done!")
```

It is highly improbable to predict the amount of time a thread could get for execution.

Example – Mutlithreading with Arguments passed to Threads

```
import threading

def print_x(x, n):
    for i in range(n):
        print(x)

if __name__ == "__main__":
    # create threads
    t1 = threading.Thread(target=print_x, args=(1, 5,))
    t2 = threading.Thread(target=print_x, args=(2, 10,))

    # start thread 1
    t1.start()
    # start thread 2
    t2.start()

    # wait until thread 1 is completely executed
    t1.join()
    # wait until thread 2 is completely executed
    t2.join()
    # both threads completely executed

    print("Done!")
```

Some More example with logging info:

```
import logging
import threading
import time
import os
def task1():
   print("Task 1 assigned to thread: {}\n".format(threading.current_thread().name))
   print("ID of process running task 1: {}\n".format(os.getpid()))

def thread_function(name):
   logging.info("Thread %s: starting\n", name)
   time.sleep(2)
   logging.info("Thread %s: finishing\n", name)

if __name__ == "__main__":
   format = "%(asctime)s: %(message)s"
   logging.basicConfig(format=format, level=logging.INFO,
                       datefmt="%H:%M:%S")

   logging.info("Main    : before creating thread\n")
   x = threading.Thread(target=thread_function, args=(1,))
   t1 = threading.Thread(target=task1, name='t1')
   logging.info("Main    : before running thread\n")
   t1.start();
   x.start()
   logging.info("Main    : wait for the thread to finish\n")
   x.join()
   t1.join()
   logging.info("Main    : all done")
```

Daemon Threads

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

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

Python waiting for the non-daemonic thread to complete. When your Python program ends, part of the shutdown process is to clean up the threading routine.

If you look at the source for Python threading, you'll see that threading._shutdown() walks through all of the running threads and calls .join() on every one that does not have the daemon flag set.
So your program waits to exit because the thread itself is waiting in a sleep. As soon as it has completed and printed the message, .join() will return and the program can exit.

x = threading.Thread(target=thread_function, args=(1,), daemon=**True**)

Signaling Between Threads through Event objects

Although the point of using multiple threads is to spin separate operations off to run concurrently, there are times when it is important to be able to synchronize the operations in two or more threads.
A simple way to communicate between threads is using Event objects.

An Event manages an internal flag that callers can either set() or clear().
Other threads can wait() for the flag to be set(), effectively blocking progress until allowed to continue.

```
import logging
import threading
import time

logging.basicConfig(level=logging.DEBUG,
                    format='(%(threadName)-10s) %(message)s',
                    )

def wait_for_event(e):
    """Wait for the event to be set before doing anything"""
    logging.debug('wait_for_event starting')
    event_is_set = e.wait()
    logging.debug('event set: %s', event_is_set)

def wait_for_event_timeout(e, t):
    """Wait t seconds and then timeout"""
    while not e.isSet():
        logging.debug('wait_for_event_timeout starting')
        event_is_set = e.wait(t)
        logging.debug('event set: %s', event_is_set)
        if event_is_set:
            logging.debug('processing event')
        else:
            logging.debug('doing other work')
```

```
e = threading.Event()
t1 = threading.Thread(name='block',
                      target=wait_for_event,
                      args=(e,))
t1.start()

t2 = threading.Thread(name='non-block',
                      target=wait_for_event_timeout,
                      args=(e, 2))
t2.start()

logging.debug('Waiting before calling Event.set()')
time.sleep(3)
e.set()
logging.debug('Event is set')
```

The wait() method takes an argument representing the number of seconds to wait for the event before timing out.
It returns a boolean indicating whether or not the event is set, so the caller knows why wait() returned. The isSet() method can be used separately on the event without fear of blocking.

In this example, wait_for_event_timeout() checks the event status without blocking indefinitely. The wait_for_event() blocks on the call to wait(), which does not return until the event status changes.

OUTPUT

```
(block     ) wait_for_event starting
(non-block ) wait_for_event_timeout starting
(MainThread) Waiting before calling Event.set()
(non-block ) event set: False
(non-block ) doing other work
(non-block ) wait_for_event_timeout starting
(MainThread) Event is set
(block     ) event set: True
(non-block ) event set: True
(non-block ) processing event_is_set
```

Race Conditions in Python

Concurrent accesses to shared resource can lead to race condition.

A race condition occurs when two or more threads can access shared data and they try to change it at the same time. As a result, the values of variables may be unpredictable and vary depending on the timings of context switches of the processes.

```
import threading
# global variable x
x = 0

def increment():
    """
    function to increment global variable x
    """
    global x
    x += 1

def thread_task():
    """
    task for thread
    calls increment function 100000 times.
    """
    for _ in range(100000):
        increment()

def main_task():
    global x
    # setting global variable x as 0
    x = 0

    # creating threads
    t1 = threading.Thread(target=thread_task)
    t2 = threading.Thread(target=thread_task)

    # start threads
    t1.start()
    t2.start()

    # wait until threads finish their job
    t1.join()
    t2.join()

if __name__ == "__main__":
    for i in range(10):
        main_task()
        print("Iteration {0}: x = {1}".format(i,x))
```

OUTPUT:

Iteration 0: x = 177753
Iteration 1: x = 168153
Iteration 2: x = 187591
Iteration 3: x = 177467
Iteration 4: x = 161099
Iteration 5: x = 164801
Iteration 6: x = 158073
Iteration 7: x = 165000
Iteration 8: x = 166891
Iteration 9: x = 155303

In above program:

Two threads t1 and t2 are created in main_task function and global variable x is set to 0.
Each thread has a target function thread_task in which increment function is called 100000 times. increment function will increment the global variable x by 1 in each call. The expected final value of x is 200000 but what we get in 10 iterations of main_task function is some different values.

This happens due to concurrent access of threads to the shared variable x. This unpredictability in value of x is nothing but race condition.

Using Locks (semaphore)

threading module provides a Lock class to deal with the race conditions**. Lock is implemented using a Semaphore object provided by the Operating System.**

A semaphore is a synchronization object that controls access by multiple processes/threads to a common resource in a parallel programming environment. It is simply a value in a designated place in operating system (or kernel) storage that each process/thread can check and then change. Depending on the value that is found, the process/thread can use the resource or will find that it is already in use and must wait for some period before trying again. Semaphores can be binary (0 or 1) or can have additional values.

Typically, a process/thread using semaphores checks the value and then, if it using the resource, changes the value to reflect this so that subsequent semaphore users will know to wait.

Lock class provides following methods:

acquire([blocking]) : To acquire a lock. A lock can be blocking or non-blocking.
When invoked with the blocking argument set to True (the default), thread execution is blocked until the lock is unlocked, then lock is set to locked and return True.When invoked with the blocking

argument set to False, thread execution is not blocked. If lock is unlocked, then set it to locked and return True else return False immediately.

release() : To release a lock.
When the lock is locked, reset it to unlocked, and return. If any other threads are blocked waiting for the lock to become unlocked, allow exactly one of them to proceed.

If lock is already unlocked, a ThreadError is raised.

```
import threading
# global variable x
x = 0

def increment():
    """
    function to increment global variable x
    """
    global x
    x += 1

def thread_task(lock):
    """
    task for thread
    calls increment function 100000 times.
    """
    for _ in range(100000):
        lock.acquire()
        increment()
        lock.release()
def main_task():
    global x
    # setting global variable x as 0
    x = 0

    # creating a lock
    lock = threading.Lock()

    # creating threads
    t1 = threading.Thread(target=thread_task, args=(lock,))
    t2 = threading.Thread(target=thread_task, args=(lock,))

    # start threads
    t1.start()
```

```
    t2.start()

    # wait until threads finish their job
    t1.join()
    t2.join()

if __name__ == "__main__":
    for i in range(10):
        main_task()
        print("Iteration {0}: x = {1}".format(i,x))
```

OUTPUT:

```
Iteration 0: x = 200000
Iteration 1: x = 200000
Iteration 2: x = 200000
Iteration 3: x = 200000
Iteration 4: x = 200000
Iteration 5: x = 200000
Iteration 6: x = 200000
Iteration 7: x = 200000
Iteration 8: x = 200000
Iteration 9: x = 200000
```

Some more intresting example with lock

To find out whether another thread has acquired the lock without holding up the current thread, pass False for the blocking argument to acquire(). In the next example, worker() tries to acquire the lock three separate times, and counts how many attempts it has to make to do so. In the mean time, lock_holder() cycles between holding and releasing the lock, with short pauses in each state used to simulate load.

```
import logging
import threading
import time

logging.basicConfig(level=logging.DEBUG,
                    format='(%(threadName)-10s) %(message)s',
                    )

def lock_holder(lock):
    logging.debug('Starting')
    while True:
```

```
        lock.acquire()
        try:
            logging.debug('Holding')
            time.sleep(0.5)
        finally:
            logging.debug('Not holding')
            lock.release()
        time.sleep(0.5)
    return

def worker(lock):
    logging.debug('Starting')
    num_tries = 0
    num_acquires = 0
    while num_acquires < 3:
        time.sleep(0.5)
        logging.debug('Trying to acquire')
        have_it = lock.acquire(0)
        try:
            num_tries += 1
            if have_it:
                logging.debug('Iteration %d: Acquired',  num_tries)
                num_acquires += 1
            else:
                logging.debug('Iteration %d: Not acquired', num_tries)
        finally:
            if have_it:
                lock.release()
    logging.debug('Done after %d iterations', num_tries)


lock = threading.Lock()

holder = threading.Thread(target=lock_holder, args=(lock,), name='LockHolder')
holder.setDaemon(True)
holder.start()

worker = threading.Thread(target=worker, args=(lock,), name='Worker')
worker.start()
```

It takes worker() more than three iterations to acquire the lock three separate times.

OUTPUT

```
$ python threading_lock_noblock.py

(LockHolder) Starting
(LockHolder) Holding
(Worker    ) Starting
(LockHolder) Not holding
(Worker    ) Trying to acquire
(Worker    ) Iteration 1: Acquired
(Worker    ) Trying to acquire
(LockHolder) Holding
(Worker    ) Iteration 2: Not acquired
(LockHolder) Not holding
(Worker    ) Trying to acquire
(Worker    ) Iteration 3: Acquired
(LockHolder) Holding
(Worker    ) Trying to acquire
(Worker    ) Iteration 4: Not acquired
(LockHolder) Not holding
(Worker    ) Trying to acquire
(Worker    ) Iteration 5: Acquired
(Worker    ) Done after 5 iterations
```

Locks as Context Managers

Locks implement the context manager API and are compatible with the with statement. Using with removes the need to explicitly acquire and release the lock.

```
import threading
import logging

logging.basicConfig(level=logging.DEBUG,
                    format='(%(threadName)-10s) %(message)s',
                    )

def worker_with(lock):
    with lock:
        logging.debug('Lock acquired via with')

def worker_no_with(lock):
    lock.acquire()
```

```
    try:
        logging.debug('Lock acquired directly')
    finally:
        lock.release()

lock = threading.Lock()
w = threading.Thread(target=worker_with, args=(lock,))
nw = threading.Thread(target=worker_no_with, args=(lock,))

w.start()
nw.start()
```

The two functions worker_with() and worker_no_with() manage the lock in equivalent ways.

OUTPUT:

```
$ python threading_lock_with.py

(Thread-1  ) Lock acquired via with
(Thread-2  ) Lock acquired directly
```

Re-entrant Locks

Normal Lock objects cannot be acquired more than once, even by the same thread. This can introduce undesirable side-effects if a lock is accessed by more than one function in the same call chain.

```
import threading

lock = threading.Lock()

print ("First try :", lock.acquire())
print ("Second try :" ,lock.acquire(0))
```

```
First try : True
Second try : False
```

In a situation where separate code from the same thread needs to “re-acquire” the lock, use an RLock instead.

```
import threading
lock = threading.RLock()
print 'First try :', lock.acquire()
print 'Second try:', lock.acquire(0)
```

First try : True
Second try : True

Produce and Consumer problem:

In addition to using Events, another way of synchronizing threads is through using a Condition object. Because the Condition uses a Lock, it can be tied to a shared resource. This allows threads to wait for the resource to be updated.

In this example, the consumer() threads wait() for the Condition to be set before continuing.
The producer() thread is responsible for setting the condition and notifying the other threads that they can continue.

```
import logging
import threading
import time

logging.basicConfig(level=logging.DEBUG,
                    format='%(asctime)s (%(threadName)-2s) %(message)s',
                    )

def consumer(cond):
    """wait for the condition and use the resource"""
    logging.debug('Starting consumer thread')
    t = threading.currentThread()
    with cond:
        cond.wait()
        logging.debug('Resource is available to consumer')

def producer(cond):
    """set up the resource to be used by the consumer"""
    logging.debug('Starting producer thread')
    with cond:
        logging.debug('Making resource available by Producer')
        cond.notifyAll()

condition = threading.Condition()
c1 = threading.Thread(name='c1', target=consumer, args=(condition,))
c2 = threading.Thread(name='c2', target=consumer, args=(condition,))
p = threading.Thread(name='p', target=producer, args=(condition,))

c1.start()
time.sleep(2)
c2.start()
```

time.sleep(2)
p.start()
The threads use with to acquire the lock associated with the Condition. Using the acquire() and release() methods explicitly also works.

OUTPUT:
$ python threading_condition.py

2013-02-21 06:37:49,549 (c1) Starting consumer thread
2013-02-21 06:37:51,550 (c2) Starting consumer thread
2013-02-21 06:37:53,551 (p) Starting producer thread
2013-02-21 06:37:53,552 (p) Making resource available by Producer
2013-02-21 06:37:53,552 (c2) Resource is available to consumer
2013-02-21 06:37:53,553 (c1) Resource is available to consumer

Threading Objects

There are a few more primitives offered by the Python threading module

Semaphore

The first Python threading object to look at is threading. Semaphore. A Semaphore is a counter with a few special properties. The first one is that the counting is atomic.
This means that there is a guarantee that the operating system will not swap out the thread in the middle of incrementing or decrementing the counter. The internal counter is incremented when you call .release() and decremented when you call .acquire().The next special property is that if a thread calls .acquire() when the counter is zero, that thread will block until a different thread calls .release() and increments the counter to one. Semaphores are frequently used to protect a resource that has a limited capacity.

An example would be if you have a pool of connections and want to limit the size of that pool to a specific number.

```
import logging
import random
import threading
import time

logging.basicConfig(level=logging.DEBUG,
                    format='%(asctime)s (%(threadName)-2s) %(message)s',
                    )

class ActivePool(object):
    def __init__(self):
        super(ActivePool, self).__init__()
        self.active = []
        self.lock = threading.Lock()
    def makeActive(self, name):
        with self.lock:
            self.active.append(name)
            logging.debug('Running: %s', self.active)
```

```
    def makeInactive(self, name):
        with self.lock:
            self.active.remove(name)
            logging.debug('Running: %s', self.active)

def worker(s, pool):
    logging.debug('Waiting to join the pool')
    with s:
        name = threading.currentThread().getName()
        pool.makeActive(name)
        time.sleep(0.1)
        pool.makeInactive(name)

pool = ActivePool()
s = threading.Semaphore(2)
for i in range(4):
    t = threading.Thread(target=worker, name=str(i), args=(s, pool))
    t.start()
```

OUTPUT:

```
2020-08-01 16:37:44,929 (0 ) Waiting to join the pool
2020-08-01 16:37:44,930 (1 ) Waiting to join the pool
2020-08-01 16:37:44,932 (2 ) Waiting to join the pool
2020-08-01 16:37:44,933 (3 ) Waiting to join the pool
2020-08-01 16:37:44,943 (0 ) Running: ['0']
2020-08-01 16:37:45,026 (1 ) Running: ['0', '1']
2020-08-01 16:37:45,127 (0 ) Running: ['1']
2020-08-01 16:37:45,134 (1 ) Running: []
2020-08-01 16:37:45,141 (2 ) Running: ['2']
2020-08-01 16:37:45,144 (3 ) Running: ['2', '3']
2020-08-01 16:37:45,244 (2 ) Running: ['3']
2020-08-01 16:37:45,252 (3 ) Running: []
```

Thread-specific Data

While some resources need to be locked so multiple threads can use them, others need to be protected so that they are hidden from view in threads that do not “own” them. The local() function creates an object capable of hiding values from view in separate threads.

```
import random
import threading
import logging

logging.basicConfig(level=logging.DEBUG,
                    format='(%(threadName)-10s) %(message)s',
                    )
```

```
def show_value(data):
    try:
        val = data.value
    except AttributeError:
        logging.debug('No value yet')
    else:
        logging.debug('value=%s', val)


def worker(data):
    show_value(data)
    data.value = random.randint(1, 100)
    show_value(data)

local_data = threading.local()
show_value(local_data)
local_data.value = 1000
show_value(local_data)

for i in range(2):
    t = threading.Thread(target=worker, args=(local_data,))
    t.start()
```

Notice that local_data.value is not present for any thread until it is set in that thread.

OUTPUT

```
$ python threading_local.py

(MainThread) No value yet
(MainThread) value=1000
(Thread-1  ) No value yet
(Thread-1  ) value=34
(Thread-2  ) No value yet
(Thread-2  ) value=7
```

Timer

A threading.Timer is a way to schedule a function to be called after a certain amount of time has passed.
You create a Timer by passing in a number of seconds to wait and a function to call:
t = threading.Timer(30.0, my_function)
You start the Timer by calling .start(). The function will be called on a new thread at some point after the specified time,
but be aware that there is no promise that it will be called exactly at the time you want.

If you want to stop a Timer that you've already started, you can cancel it by calling .cancel().

Calling .cancel() after the Timer has triggered does nothing and does not produce an exception.
A Timer can be used to prompt a user for action after a specific amount of time.
If the user does the action before the Timer expires, .cancel() can be called.

Timer Objects
This class represents an action that should be run only after a certain amount of time has passed — a timer.
Timer is a subclass of Thread and as such also functions as an example of creating custom threads.

Timers are started, as with threads, by calling their start() method. The timer can be stopped (before its action has begun) by calling the cancel() method.
The interval the timer will wait before executing its action may not be exactly the same as the interval specified by the user.

For example:

```
def hello():
   print("hello, world")

t = Timer(30.0, hello)
t.start()  # after 30 seconds, "hello, world" will be printed
```

Barrier

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

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

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

Python RegEx

examples:

^a...s$

The pattern is: any five letter string starting with a and ending with s
Expression String Matched?

^a...s$	abs		No match
		alias	Match
		abyss	Match
		Alias	No match
		An abacus	No match

```
import re

pattern = '^a...s$'
test_string = 'abyss'
result = re.match(pattern, test_string)

if result:
  print("Search successful.")
else:
  print("Search unsuccessful.")
```

Specify Pattern Using RegEx

MetaCharacters
Metacharacters are characters that are interpreted in a special way by a RegEx engine. Here's a list of metacharacters:

[] . ^ $ * + ? {} () \ |

Metacharacters Supported by the re Module

The following table briefly summarizes all the metacharacters supported by the re module. Some characters serve more than one purpose:

Character(s)	Meaning
.	Matches any single character except newline
^	· Anchors a match at the start of a string · Complements a character class
$	Anchors a match at the end of a string
*	Matches zero or more repetitions
+	Matches one or more repetitions
?	· Matches zero or one repetition · Specifies the non-greedy versions of *, +, and ? · Introduces a lookahead or lookbehind assertion · Creates a named group
{}	Matches an explicitly specified number of repetitions
\	· Escapes a metacharacter of its special meaning

· Introduces a special character class
· Introduces a grouping backreference
[] Specifies a character class
| Designates alternation
() Creates a group

:
#
=
! Designate a specialized group

<> Creates a named group

[] - Square brackets

Square brackets specify a set of characters you wish to match.

Expression	String	Matched?
[abc]	a	1 match
ac	2	matches
Hey Jude		No match
abc de ca	5	matches

You can also specify a range of characters using - inside square brackets.

[a-e] is the same as [abcde].
[1-4] is the same as [1234].
[0-39] is the same as [01239].
You can complement (invert) the character set by using caret ^ symbol at the start of a square-bracket.

[^abc] means any character except a or b or c.
[^0-9] means any non-digit character.

re.search('ba[artz]', 'foobarqux')
<_sre.SRE_Match object; span=(3, 6), match='bar'>

>>> re.search('[0-9a-fA-f]', '--- a0 ---')
<_sre.SRE_Match object; span=(4, 5), match='a'>

re.search() scans the search string from left to right, and as soon as it locates a match for <regex>, it stops scanning and returns the match.

. - Period

A period matches any single character (except newline '\n').

Expression	String	Matched?
..	a	No match
ac	1	match
acd	1	match
acde	2	matches (contains 4 characters)

<_sre.SRE_Match object; span=(0, 7), match='fooxbar'>

```
>>> print(re.search('foo.bar', 'foobar'))
None
>>> print(re.search('foo.bar', 'foo\nbar'))
None
```

^ - Caret

The caret symbol ^ is used to check if a string starts with a certain character.

Expression	String	Matched?
^a	a	1 match
	abc	1 match
	bac	No match
^ab	abc	1 match
	acb	No match (starts with a but not followed by b)

$ - Dollar

The dollar symbol $ is used to check if a string ends with a certain character.

Expression	String	Matched?
a$	a	1 match
	formula	1 match
	cab	No match

*** - Star**

The star symbol * matches zero or more occurrences of the pattern left to it.

Expression	**String**	**Matched?**
ma*n	mn	1 match
	man	1 match
	maaan	1 match
	main	No match (a is not followed by n)
	woman	1 match

\+ - **Plus**
The plus symbol + matches **one or more occurrences** of the pattern left to it.

Expression	**String**	**Matched?**
ma+n	mn	No match (no a character)
	man	1 match
	maaan	1 match
	main	No match (a is not followed by n)
	woman	1 match

\w
\W

Match based on whether a character is a word character.

```
import re
txt = "The rain in Spain"
#Return a match at every NON word character (characters NOT between a and Z. Like "!", "?" white-space etc.):
x = re.findall("\W", txt)
print(x)
if x:
  print("Yes, there is at least one match!")
else:
  print("No match")
```

output:

```
 [' ', ' ', ' ']
Yes, there is at least one match!
```

Some more

```
>>> re.search('\w', '#(.a$@&')
<_sre.SRE_Match object; span=(3, 4), match='a'>
```

```
>>> re.search('[a-zA-Z0-9_]', '#(.a$@&')
<_sre.SRE_Match object; span=(3, 4), match='a'>
```

\d
\D

Match based on whether a character is a decimal digit.

```
>> re.search('\d', 'abc4def')
<_sre.SRE_Match object; span=(3, 4), match='4'>
>>> re.search('\D', '234Q678')
<_sre.SRE_Match object; span=(3, 4), match='Q'>
```

? - Question Mark

The question mark symbol ? matches zero or one occurrence of the pattern left to it.

Expression	String	Matched?
ma?n	mn	1 match
	man	1 match
	maaan	No match (more than one a character)
	main	No match (a is not followed by n)
	woman	1 match

{} - Braces

Consider this code: {n,m}. This means at least n, and at most m repetitions of the pattern left to it.

Expression	String	Matched?
a{2,3}	abc dat	No match
	abc daat	1 match (at daat)
	aabc daaat	2 matches (at aabc and daaat)
	aabc daaaat	2 matches (at aabc and daaaat)

Let's try one more example. This RegEx [0-9]{2, 4} matches at least 2 digits but not more than 4 digits

Expression	String	Matched?
[0-9]{2,4}	ab123csde	1 match (match at ab123csde)
	12 and 345673	3 matches (12, 3456, 73)
	1 and 2	No match

| - Alternation

Vertical bar | is used for alternation (or operator).

Expression	String	Matched?	
a|b		cde	No match
		ade	1 match (match at ade)
		acdbea	3 matches (at acdbea)

Here, a|b match any string that contains either a or b

() - Group

Parentheses () is used to group sub-patterns. For example, (a|b|c)xz match any string that matches either a or b or c followed by xz

Expression	String	Matched?	
(a|b|c)xz	ab xz		No match
		abxz	1 match (match at abxz)
		axz cabxz	2 matches (at axzbc cabxz)

\d is essentially equivalent to [0-9], and \D is equivalent to [^0-9].

\ - Backslash

Backlash \ is used to escape various characters including all metacharacters. For example,

\$a match if a string contains $ followed by a. Here, $ is not interpreted by a RegEx engine in a special way. If you are unsure if a character has special meaning or not, you can put \ in front of it. This makes sure the character is not treated in a special way.

\s
\S

Match based on whether a character represents whitespace.

\s matches any whitespace character:

```
>>> re.search('\s', 'foo\nbar baz')
<_sre.SRE_Match object; span=(3, 4), match='\n'>
```
Note that, unlike the dot wildcard metacharacter, \s does match a newline character.

\S is the opposite of \s. It matches any character that isn't whitespace:

```
>>> re.search('\S', '  \n foo  \n  ')
<_sre.SRE_Match object; span=(4, 5), match='f'>
```
Again, \s and \S consider a newline to be whitespace. In the example above, the first non-whitespace character is 'f'.

The character class sequences \w, \W, \d, \D, \s, and \S can appear inside a square bracket character class as well:

```
>>> re.search('[\d\w\s]', '---3---')
<_sre.SRE_Match object; span=(3, 4), match='3'>
>>> re.search('[\d\w\s]', '---a---')
<_sre.SRE_Match object; span=(3, 4), match='a'>
>>> re.search('[\d\w\s]', '--- ---')
<_sre.SRE_Match object; span=(3, 4), match=' '>
```
In this case, [\d\w\s] matches any digit, word, or whitespace character.

Escaping Metacharacters

backslash (\)

Removes the special meaning of a metacharacter.

```
>>> re.search('.', 'foo.bar')
<_sre.SRE_Match object; span=(0, 1), match='f'>

>>> re.search('\.', 'foo.bar')
<_sre.SRE_Match object; span=(3, 4), match='.'>
```
In the <regex> on line 1, the dot (.) functions as a wildcard metacharacter, which matches the first character in the string ('f').
The . character in the <regex> on line 4 is escaped by a backslash, so it isn't a wildcard.
It's interpreted literally and matches the '.' at index 3 of the search string.

Using backslashes for escaping can get messy. Suppose you have a string that contains a single backslash:

```
>>> s = r'foo\bar'
>>> print(s)
```

foo\bar

Now suppose you want to create a <regex> that will match the backslash between 'foo' and 'bar'.
The backslash is itself a special character in a regex, so to specify a literal backslash, you need to escape it with another backslash. If that's that case, then the following should work:

```
>>> re.search('\\', s)
```

This will throw lots of error messages.

The problem here is that the backslash escaping happens twice, first by the Python interpreter on the string literal and then again by the regex parser on the regex it receives.

Here's the sequence of events:

The Python interpreter is the first to process the string literal '\\'.
It interprets that as an escaped backslash and passes only a single backslash to re.search().
The regex parser receives just a single backslash, which isn't a meaningful regex, so the messy error ensues.

here are two ways around this. First, you can escape both backslashes in the original string literal:

```
>>> re.search('\\\\', s)
<_sre.SRE_Match object; span=(3, 4), match='\\'>
```

Doing so causes the following to happen:

The interpreter sees '\\\\' as a pair of escaped backslashes. It reduces each pair to a single backslash and passes '\\' to the regex parser.
The regex parser then sees \\ as one escaped backslash. As a <regex>, that matches a single backslash character.
You can see from the match object that it matched the backslash at index 3 in s as intended. It's cumbersome, but it works.
The second, and probably cleaner, way to handle this is to specify the <regex> using a raw string:

```
>>> re.search(r'\\', s)
<_sre.SRE_Match object; span=(3, 4), match='\\'>
```

Anchors

Anchors are zero-width matches. They don’t match any actual characters in the search string, and they don’t consume any of the search string during parsing. Instead, an anchor dictates a particular location in the search string where a match must occur.

^
\A

Anchor a match to the start of <string>.

When the regex parser encounters ^ or \A, the parser’s current position must be at the beginning of the search string for it to find a match.

```
>>> re.search('^foo', 'foobar')
<_sre.SRE_Match object; span=(0, 3), match='foo'>
>>> print(re.search('^foo', 'barfoo'))
None
```

\A functions similarly:

```
>>> re.search('\Afoo', 'foobar')
<_sre.SRE_Match object; span=(0, 3), match='foo'>
>>> print(re.search('\Afoo', 'barfoo'))
None
```

and \A behave slightly differently from each other in MULTILINE mode. You’ll learn more about MULTILINE mode below in the section on flags.

$
\Z

Anchor a match to the end of <string>.

When the regex parser encounters $ or \Z, the parser’s current position must be at the end of the search string for it to find a match. Whatever precedes $ or \Z must constitute the end of the search string:

```
>>> re.search('bar$', 'foobar')
<_sre.SRE_Match object; span=(3, 6), match='bar'>
>>> print(re.search('bar$', 'barfoo'))
None

>>> re.search('bar\Z', 'foobar')
<_sre.SRE_Match object; span=(3, 6), match='bar'>
>>> print(re.search('bar\Z', 'barfoo'))
```

None

$ and \Z behave slightly differently from each other in MULTILINE mode. See the section below on flags for more information on MULTILINE mode.

\b

Anchors a match to a word boundary.
\b - Matches if the specified characters are at the beginning or end of a word.
example:

```
>>> re.search(r'\bbar', 'foo bar')
<_sre.SRE_Match object; span=(4, 7), match='bar'>
>>> re.search(r'\bbar', 'foo.bar')
<_sre.SRE_Match object; span=(4, 7), match='bar'>

>>> print(re.search(r'\bbar', 'foobar'))
None

>>> re.search(r'foo\b', 'foo bar')
<_sre.SRE_Match object; span=(0, 3), match='foo'>
>>> re.search(r'foo\b', 'foo.bar')
<_sre.SRE_Match object; span=(0, 3), match='foo'>

>>> print(re.search(r'foo\b', 'foobar'))
None
```

\B - Opposite of \b. Matches if the specified characters are not at the beginning or end of a word.

```
>>> print(re.search(r'\Bfoo\B', 'foo'))
None
>>> print(re.search(r'\Bfoo\B', '.foo.'))
None

>>> re.search(r'\Bfoo\B', 'barfoobaz')
<_sre.SRE_Match object; span=(3, 6), match='foo'>
```

\d - Matches any decimal digit. Equivalent to [0-9]
\D - Matches any non-decimal digit. Equivalent to [^0-9]

\s - Matches where a string contains any whitespace character. Equivalent to [\t\n\r\f\v].
\S - Matches where a string contains any non-whitespace character. Equivalent to [^ \t\n\r\f\v].

\Z - Matches if the specified characters are at the end of a string.

The regex <.*> effectively means:

A '<' character
Then any sequence of characters
Then a '>' character

But which '>' character? There are three possibilities:

The one just after 'foo'
The one just after 'bar'
The one just after 'baz'
Since the * metacharacter is greedy, it dictates the longest possible match, which includes everything up to and including the '>' character
that follows 'baz'. You can see from the match object that this is the match produced.

If you want the shortest possible match instead, then use the non-greedy metacharacter sequence *?:

```
>>> re.search('<.*?>', '%<foo> <bar> <baz>%')
<_sre.SRE_Match object; span=(1, 6), match='<foo>'>
```
In this case, the match ends with the '>' character following 'foo'.

Note: You could accomplish the same thing with the regex <[^>]*>, which means:

A '<' character
Then any sequence of characters other than '>'
Then a '>' character
This is the only option available with some older parsers that don't support lazy quantifiers.
Happily, that's not the case with the regex parser in Python's re module.

{m}

Matches exactly m repetitions of the preceding regex.

None

```
>>> re.search('x-{3}x', 'x---x')                # Three dashes
<_sre.SRE_Match object; span=(0, 5), match='x---x'>

>>> print(re.search('x-{3}x', 'x----x'))         # Four dashes
```

None

{m,n}

Matches any number of repetitions of the preceding regex from m to n, inclusive.

<regex>)

Defines a subexpression or group.
>>> re.search('(bar)', 'foo bar baz')
<_sre.SRE_Match object; span=(4, 7), match='bar'>

Capturing Groups
Grouping isn't the only useful purpose that grouping constructs serve. Most (but not quite all) grouping constructs also capture the
part of the search string that matches the group. You can retrieve the captured portion or refer to it later in several different ways.

Remember the match object that re.search() returns? There are two methods defined for a match object that provide access to
captured groups: .groups() and .group().

m.groups()

Returns a tuple containing all the captured groups from a regex match.

Consider this example:

>>> m = re.search('(\w+),(\w+),(\w+)', 'foo,quux,baz')
>>> m
<_sre.SRE_Match object; span=(0, 12), match='foo:quux:baz'>

Each of the three (\w+) expressions matches a sequence of word characters. The full regex (\w+),(\w+),(\w+) breaks the search string into
three comma-separated tokens.

Deep Dive: Debugging Regular Expression Parsing

As you know from above, the metacharacter sequence {m,n} indicates a specific number of repetitions. It matches anywhere from m to n repetitions of what precedes it:

```
>>> re.search('x[123]{2,4}y', 'x222y')
<_sre.SRE_Match object; span=(0, 5), match='x222y'>
```
You can verify this with the DEBUG flag:

```
>>> re.search('x[123]{2,4}y', 'x222y', re.DEBUG)
LITERAL 120
MAX_REPEAT 2 4
  IN
    LITERAL 49
    LITERAL 50
    LITERAL 51
LITERAL 121
<_sre.SRE_Match object; span=(0, 5), match='x222y'>
```
MAX_REPEAT 2 4 confirms that the regex parser recognizes the metacharacter sequence {2,4} and interprets it as a range quantifier.

But, as noted previously, if a pair of curly braces in a regex in Python contains anything other than a valid number or numeric range,
then it loses its special meaning.

You can verify this also:

```
>>> re.search('x[123]{foo}y', 'x222y', re.DEBUG)
LITERAL 120
IN
  LITERAL 49
  LITERAL 50
  LITERAL 51
LITERAL 123
LITERAL 102
LITERAL 111
LITERAL 111
LITERAL 125
LITERAL 121
```

The findall() Function
The findall() function returns a list containing all matches.

Example
Print a list of all matches:

```
import re
```

```
txt = "The rain in Spain"
x = re.findall("ai", txt)
print(x)
```

Return an empty list if no match was found:

```
import re

txt = "The rain in Spain"
x = re.findall("Portugal", txt)
print(x)
```

re.findall()

```
# Program to extract numbers from a string
import re
string = 'hello 12 hi 89. Howdy 34'
pattern = '\d+'
result = re.findall(pattern, string)
print(result)
# Output: ['12', '89', '34']
```

The search() Function

The search() function searches the string for a match, and returns a Match object if there is a match.

If there is more than one match, only the first occurrence of the match will be returned:

re.search()

```
import re
string = "Python is fun"
# check if 'Python' is at the beginning
match = re.search('\APython', string)
if match:
  print("pattern found inside the string")
else:
  print("pattern not found")

# Output: pattern found inside the string
```

Example

Search for the first white-space character in the string:

```
import re

txt = "The rain in Spain"
x = re.search("\s", txt)

print("The first white-space character is located in position:", x.start())
```

If no matches are found, the value None is returned:

The split() Function
The re.split method splits the string where there is a match and returns a list of strings where the splits have occurred.

Example
Split at each white-space character:

```
import re
#Split the string at the first white-space character:
txt = "The rain in Spain"
x = re.split("\s", txt)
print(x)
```

You can control the number of occurrences by specifying the maxsplit parameter:

Output:
['The', 'rain', 'in', 'Spain']

re.split()

```
import re
string = 'Twelve:12 Eighty nine:89.'
pattern = '\d+'
result = re.split(pattern, string)
print(result)
# Output: ['Twelve:', ' Eighty nine:', '.']

import re
string = 'Twelve:12 Eighty nine:89 Nine:9.'
pattern = '\d+'
# maxsplit = 1
# split only at the first occurrence
result = re.split(pattern, string, 1)
print(result)
# Output: ['Twelve:', ' Eighty nine:89 Nine:9.']
```

The sub() Function
The sub() function replaces the matches with the text of your choice:

Example
Replace every white-space character with the number 9:

```
import re
#Replace all white-space characters with the digit "9":

txt = "The rain in Spain"
x = re.sub("\s", "9", txt)
print(x)
```

You can control the number of replacements by specifying the count parameter:

output:
The9rain9in9Spain

re.sub()

```
# Program to remove all whitespaces
import re
# multiline string
string = 'abc 12\
de 23 \n f45 6'
# matches all whitespace characters
pattern = '\s+'
# empty string
replace = ''
new_string = re.sub(pattern, replace, string)
print(new_string)
# Output: abc12de23f456
```

Example
Replace the first 2 occurrences:

```
import re
#Replace the first two occurrences of a white-space character with the digit 9:
txt = "The rain in Spain"
x = re.sub("\s", "9", txt, 2)
print(x)
```

output:

The9rain9in Spain

```
import re
# multiline string
string = 'abc 12\
de 23 \n f45 6'
# matches all whitespace characters
pattern = '\s+'
replace = ''

new_string = re.sub(r'\s+', replace, string, 1)
print(new_string)

# Output:
# abc12de 23
# f45 6
```

re.subn()

```
# Program to remove all whitespaces
import re
# multiline string
string = 'abc 12\
de 23 \n f45 6'
# matches all whitespace characters
pattern = '\s+'
# empty string
replace = ''

new_string = re.subn(pattern, replace, string)
print(new_string)

# Output: ('abc12de23f456', 4)
```

Match Object

A Match Object is an object containing information about the search and the result.
Note: If there is no match, the value None will be returned, instead of the Match Object.

```
import re
#The search() function returns a Match object:

txt = "The rain in Spain"
x = re.search("ai", txt)
print(x)
```

output:
<_sre.SRE_Match object; span=(5, 7), match='ai'>

```
import re
string = '39801 356, 2102 1111'
# Three digit number followed by space followed by two digit number
pattern = '(\d{3}) (\d{2})'
# match variable contains a Match object.
match = re.search(pattern, string)
if match:
  print(match.group())
else:
  print("pattern not found")

# Output: 801 35
```

The Match object has properties and methods used to retrieve information about the search, and the result:

.span() returns a tuple containing the start-, and end positions of the match.
.string returns the string passed into the function
.group() returns the part of the string where there was a match

```
import re

#Search for an upper case "S" character in the beginning of a word, and print its position:

txt = "The rain in Spain"
x = re.search(r"\bS\w+", txt)
print(x.span())
```

output:

(12, 17)

```
import re

#The string property returns the search string:

txt = "The rain in Spain"
x = re.search(r"\bS\w+", txt)
print(x.string)
```

OUTPUT:
The rain in Spain

Example
Print the string passed into the function:

```
import re

#The string property returns the search string:

txt = "The rain in Spain"
x = re.search(r"\bS\w+", txt)
print(x.string)
```

output:
The rain in Spain

Example
Print the part of the string where there was a match.

The regular expression looks for any words that starts with an upper case "S":

```
import re

#Search for an upper case "S" character in the beginning of a word, and print the word:

txt = "The rain in Spain"
x = re.search(r"\bS\w+", txt)
print(x.group())
```

OUTPUT:

Spain

FAQS:

Functions can return multiple values

In Python, a function can return more than one value as a tuple. Take a look at the following code:
facts about python language

```
def myFunc():
   return "hara",46,50000;

name,age,sal=myFunc();
print (name,age,sal);
```

Output:

hara 46 50000

This isn't possible in a language like Java. There, you can return an array of values instead.

Python supports multiple assignments in one statement

```
a,b,c=11,22,33;

Print (a,b,c)
```

OUTPUT:

11,22,33

With slicing, it's easier to reverse a list

If we slice a list of values from starting to end but with a step of -1, we get the list right to left (reversed).

```
nums=[11,22,33,44,55,66]
nums=[::-]
print(nums)
nums[::-1]
[66, 55, 44, 33, 22, 11]
```

When is the else part of a try-except block executed?

In an if-else block, the else part is executed when the condition in the if-statement is False.
But with a try-except block, the else part executes only if no exception is raised in the try part.

If a list is nums=[0,1,2,3,4], what is nums[-1]?
This code does not throw an exception. nums[-1] is 4 because it begins traversing from right.

What is the PYTHONPATH variable?

PYTHONPATH is the variable that tells the interpreter where to locate the module files imported into a program. Hence, it must include the Python source library directory and the directories containing Python source code. You can manually set PYTHONPATH, but usually, the Python installer will preset it.

Explain join() and split() in Python.

join() lets us join characters from a string together by a character we specify.

```
>>> ','.join('12345')
‘1,2,3,4,5’
```

split() lets us split a string around the character we specify.

```
>>> '1,2,3,4,5'.split(',')
[‘1’, ‘2’, ‘3’, ‘4’, ‘5’]
```

Explain the output of the following piece of code-

```
x=[‘ab’,’cd’]
print(len(map(list,x)))
```

This actually gives us an error- a TypeError. This is because map() has no len() attribute in their dir().
Explain a few methods to implement Functionally Oriented Programming in Python.
Sometimes, when we want to iterate over a list, a few methods come in handy.

filter()

Filter lets us filter in some values based on conditional logic.

```
>>> list(filter(lambda x:x>5,range(8)))
[6, 7]
```

map()

Map applies a function to every element in an iterable.

```
>>> list(map(lambda x:x**2,range(8)))
[0, 1, 4, 9, 16, 25, 36, 49]
```

reduce()

Reduce repeatedly reduces a sequence pair-wise until we reach a single value.

```
>>> from functools import reduce
>>> reduce(lambda x,y:x-y,[1,2,3,4,5])
-13
```

So what is the output of the following piece of code?

```
x=[‘ab’,’cd’]
print(len(list(map(list,x))))
```

This outputs 2 because the length of this list is 2. list(map(list,x)) is [[‘a’, ‘b’], [‘c’, ‘d’]], and the length of this is 2.

Is del the same as remove()? What are they?

del and remove() are methods on lists/ ways to eliminate elements.

```
>>> list=[3,4,5,6,7]
>>> del list[3]
>>> list
[3, 4, 5, 7]

>>> list.remove(5)
>>> list
```

remove : remove() removes the first matching value or object, not a specific indexing. lets say list.remove(value)

del : del removes the item at a specific index. lets say del list[index]

[3, 4, 7]

How do you open a file for writing?

Let’s create a text file on our Desktop and call it tabs.txt. To open it to be able to write to it, use the following line of code-

```
>>> file=open('tabs.txt','w')
```

This opens the file in writing mode. You should close it once you’re done.

```
>>> file.close()
```

or in below way:
By using "with" statement is the safest way to handle a file operation in Python because "with" statement ensures that the file is closed when the block inside with is exited.
example:

```
with open("my_file.txt", "r") as my_file:
  # do some file operations
```

In the above example, you don't need to explicitly call the close() method. It is done internally.

Explain the output of the following piece of code-

```
>>> tuple=(123,'John')
>>> tuple*=2
>>> tuple
(123, ‘John’, 123, ‘John’)
```

In this code, we multiply the tuple by 2. This duplicates its contents, hence, giving us (123, ‘John’, 123, ‘John’).
We can also do this to strings:

```
>>> 'ba'+'na'*2
‘banana’
```

Differentiate between the append() and extend() methods of a list.

The methods append() and extend() work on lists. While append() adds an element to the end of the list, extend adds another list to the end of a list.
Let’s take two lists.

```
>>> list1,list2=[1,2,3],[5,6,7,8]
```

This is how append() works:

```
>>> list1.append(4)
>>> list1
[1, 2, 3, 4]
```

And this is how extend() works:

```
>>> list1.extend(list2)
>>> list1
```

[1, 2, 3, 4, 5, 6, 7, 8]

What does the map() function do?

map() executes the function we pass to it as the first argument; it does so on all elements of the iterable in the second argument.
Let's take an example, shall we?

```
>>> for i in map(lambda i:i**3, (2,3,7)):
print(i)
8
27
343
```
This gives us the cubes of the values 2, 3, and 7.

Explain try, raise, and finally.

These are the keywords we use with exception-handling. We put risky code under a try block, use the raise statement to explicitly raise an error, and use the finally block to put code that we want to execute anyway, irrespective of exception.

What happens if we do not handle an error in the except block?
If we don't do this, the program terminates. Then, it sends an execution trace to sys.stderr.

Is there a way to remove the last object from a list?

Yes, there is. Try running the following piece of code-

```
>>> list=[1,2,3,4,5]
>>> list.pop(-1)
5

>>> list
[1, 2, 3, 4]
```

How will you convert an integer to a Unicode character?

This is simple. All we need is the chr(x) built-in function. See how.

```
>>> chr(52)
'4'
>>> chr(49)
'1'
>>> chr(67)
```

‘C’

Explain the problem with the following piece of code-

```
>>> def func(n=[]):
        #playing around
         pass
>>> func([1,2,3])
>>> func()
>>> n
```

The request for n raises a **NameError**. This is since n is a variable local to func and we cannot access it elsewhere. It is also true that Python only evaluates default parameter values once; every invocation shares the default value. If one invocation modifies it, that is what another gets. This means you should only ever use primitives, strings, and tuples as default parameters, not mutable objects.

What do you see below?

s = a + ‘[‘ + b + ‘:’ + c + ‘]’

This is string concatenation. If a, b, and c are strings themselves, then it works fine and concatenates the strings around the strings ‘[‘, ‘:’, and ‘]’ as mentioned. If even one of these isn’t a string, this raises a TypeError.

```
a='A'

b='B'

c='C'

s = a + '[' + b + ':' + c + ']'

print(s)
```

Output: A[B:C]

So does recursion cause any trouble?

Sure does:

1) Needs more function calls.
2) Each function call stores a state variable to the program stack- consumes memory, can cause memory overflow.
3) Calling a function consumes time.

What good is recursion?

With recursion, we observe the following:

Need to put in less efforts.

Smaller code than that by loops.

Easier-to-understand code.

What does the following code give us?

>>> b=(1)

Not a tuple. This gives us a plain integer.

>>> type(b)

<class ‘int’>

To let it be a tuple, we can declare so explicitly with a comma after 1:

>>> b=(1,)

>>> type(b)

<class ‘tuple’>

Why are identifier names with a leading underscore disparaged?

Since Python does not have a concept of private variables, it is a convention to use leading underscores to declare a variable private. This is why we mustn’t do that to variables we do not want to make private.

Can you remove the whitespaces from the string “aaa bbb ccc ddd eee”?

I can think of three ways to do this.

Using join-

>>> s='aaa bbb ccc ddd eee'

>>> s1=''.join(s.split())

>>> s1

‘aaabbbcccdddeee’

Using a list comprehension–

```
>>> s='aaa bbb ccc ddd eee'
>>> s1=str(''.join(([i for i in s if i!=' '])))
>>> s1
'aaabbbcccdddeee'
```

Using replace()-

```
>>> s='aaa bbb ccc ddd eee'
>>> s1 = s.replace(' ','')
>>> s1
'aaabbbcccdddeee'
```

How do you get the current working directory using Python?

Working on software with Python, you may need to read and write files from various directories. To find out which directory we're presently working under, we can borrow the getcwd() method from the os module.

```
>>> import os
>>> os.getcwd()
'C:\\Users\\haramohan.sahu\\Desktop\\PhthonDoc'
```

For this, we'll import the function shuffle() from the module random.

```
mylist=[ declare these values]
>>> from random import shuffle
>>> shuffle(mylist)
>>> mylist
[3, 4, 8, 0, 5, 7, 6, 2, 1]
```

How do you remove the leading whitespace in a string?

Leading whitespace in a string is the whitespace in a string before the first non-whitespace character. To remove it from a string, we use the method lstrip().

```
>>> ' Ayushi '.lstrip()
```

‘Ayushi ‘

As you can see, this string had both leading and trailing whitespaces. lstrip() stripped the string of the leading whitespace.

If we want to strip the trailing whitespace instead, we use rstrip().

```
>>> ' Ayushi '.rstrip()
```

‘ Ayushi’

Below, we give you code to remove numbers smaller than 5 from the list nums. However, it does not work as expected. Can you point out the bug for us?

```
>>> nums=[1,2,5,10,3,100,9,24]
>>> for i in nums:
        if i<5:
            nums.remove(i)
>>> nums
[2, 5, 10, 100, 9, 24]
```

This code checks for each element in nums- is it smaller than 5? If it is, it removes that element. In the first iteration, 1 indeed is smaller than 5. So it removes that from this list. But this disturbs the indices. Hence, it checks the element 5, but not the element 2. For this situation, we have three workarounds:

Create an empty array and append to that-

```
>>> nums=[1,2,5,10,3,100,9,24]
>>> newnums=[]
>>> for i in nums:
    if i>=5:
```

```
newnums.append(i)
>>> newnums
[5, 10, 100, 9, 24]
```

Using a list comprehension-

```
>>> nums=[1,2,5,10,3,100,9,24]
>>> newnums=[i for i in nums if i>=5]
>>> newnums
[5, 10, 100, 9, 24]
```

Using the filter() function-

```
>>> nums=[1,2,5,10,3,100,9,24]
>>> newnums=list(filter(lambda x:x>=5, nums))
>>> newnums
[5, 10, 100, 9, 24]
```

What is the enumerate() function in Python?

enumerate() iterates through a sequence and extracts the index position and its corresponding value too.

Let's take an example.

```
>>> for i,v in enumerate(['Python','C++','Scala']):
        print(i,v)
0 Python
1 C++
2 Scala
```

How will you create the following pattern using Python?

```
*
**
```

***

****

*****

We will use two for-loops for this.

```
>>> for i in range(1,6):
    for j in range(1,i+1):
        print('*',end='')
```

Where will you use while rather than for?

Although we can do with for all that we can do with while, there are some places where a while loop will make things easier-

->For simple repetitive looping

->When we don't need to iterate through a list of items- like database records and characters in a string.

Take a look at this piece of code:

```
>>> A0= dict(zip(('a','b','c','d','e'),(1,2,3,4,5)))
>>> A1= range(10)
>>> A2= sorted([i for i in A1 if i in A0])
>>> A3= sorted([A0[s] for s in A0])
>>> A4= [i for i in A1 if i in A3]
>>> A5= {i:i*i for i in A1}
>>> A6= [[i,i*i] for i in A1]
>>> A0,A1,A2,A3,A4,A5,A6
```

What are the values of variables A0 to A6? Explain.

Here you go:

A0= {‘a’: 1, ‘b’: 2, ‘c’: 3, ‘d’: 4, ‘e’: 5}

A1= range(0, 10)

A2= []

A3= [1, 2, 3, 4, 5]

A4= [1, 2, 3, 4, 5]

A5= {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}

A6= [[0, 0], [1, 1], [2, 4], [3, 9], [4, 16], [5, 25], [6, 36], [7, 49], [8, 64], [9, 81]]

Now to find out what happened. A0 zips ‘a’ with 1, ‘b’ with 2, and so on. This results in tuples, which the call to dict()

then turns into a dictionary by using these as keys and values.

A1 gives us a range object with start=0 and stop=10.

A2 checks each item in A1- does it exist in A0 as well? If it does, it adds it to a list. Finally, it sorts this list.

Since no items exist in both A0 and A1, this gives us an empty list.

A3 takes each key in A0 and returns its value. This gives us the list [1,2,3,4,5].

A4 checks each item in A1- does it exist in A3 too? If it does, it adds it to a list and returns this list.

A5 takes each item in A1, squares it, and returns a dictionary with the items in A1 as keys and their squares as the corresponding values.

A6 takes each item in A1, then returns sublists containing those items and their squares- one at a time.

Does Python have a switch-case statement?

In Python, we do not have a switch-case statement. Here, you may write a switch function to use. Else, you may use a set of if-elif-else statements. To implement a function for this, we may use a dictionary.

```
>>> def switch(choice):
  switcher={
     'Ayushi':'Monday',
     'Megha':'Tuesday',
    print(switcher.get(choice,'Hi, user'))
```

return

>>> switch('Megha')

Tuesday

>>> switch('Ayushi')

Monday

>>> switch('Ruchi')

Hi, user

Here, the get() method returns the value of the key. When no key matches, the default value (the second argument) is returned.

Differentiate between deep and shallow copy.

A deep copy copies an object into another. This means that if you make a change to a copy of an object, it won't affect the original object.

In Python, we use the function deepcopy() for this, and we import the module copy. We use it like:

>>> import copy

>>> b=copy.deepcopy(a)

A shallow copy, however, copies one object's reference to another. So, if we make a change in the copy, it will affect the original object.

For this, we have the function copy(). We use it like:

b=copy.copy(a)

Python Developer Interview Questions

Can you make a local variable's name begin with an underscore? (developer)

You can, but you should not. This is because:

Local variables indicate private variables of a class, and so, they confuse the interpreter.

Is a NumPy array better than a list?

NumPy arrays have 3 benefits over lists:

- They are faster
- They require less memory
- They are more convenient to work with

If you installed a module with pip but it doesn't import in your IDLE, what could it possibly be?

Well, for one, it could be that I installed two versions of Python on my system- possibly, both 32-bit and 64-bit.

The Path variable in my system's environment variables is probably set to both, but one of them prior to the other- say, the 32-bit.

This made the command prompt use the 32-bit version of pip to install the module I chose.

When I ran the IDLE, I ran the 64-bit version.

As this sequence of events unlapped, I couldn't import the module I just installed.

I could do two things.

The temporary solution- I will add the path to sys manually every time I work on a new session of the interpreter.

>>> sys.path.append('C:\\Users\\Ayushi\\AppData\\Local\\Programs\\Python\\Python37\\Scripts')

The permanent solution- I will update the value of Path in my environment variables to hold the location of the Scripts folder for the 64-bit version first.

If while installing a package with pip, you get the error No matching installation found, what can you do?

In such a situation, one thing I can do is to download the binaries for that package from the following location:

https://www.lfd.uci.edu/~gohlke/pythonlibs/

Then, I can install the wheel using pip.

How do you debug a program in Python? Answer in brief.

To debug a Python program, we use the pdb module. This is the Python debugger. If we start a program using pdb, it will let us step through the code.

Python OOPS and Library Interview Questions

Can I dynamically load a module in Python?

Dynamic loading is where we do not load a module till we need it. This is slow, but lets us utilize the memory more efficiently. In Python, you can use the importlib module for this:

```
import importlib

module = importlib.import_module('my_package.my_module')
```

Which methods/functions do we use to determine the type of instance and inheritance?

Here, we talk about three methods/functions- type(), isinstance(), and issubclass().

a. type()

This tells us the type of object we're working with.

>>> type(3)

<class 'int'>

>>> type(False)

<class 'bool'>

>>> type(lambda :print("Hi"))

<class 'function'>

>>> type(type)

<class 'type'>

b. isinstance()

This takes in two arguments- a value and a type. If the value is of the kind of the specified type, it returns True. Else, it returns False.

>>> isinstance(3,int)

True

>>> isinstance((1),tuple)

False

>>> isinstance((1,),tuple)

True

c. issubclass()

This takes two classes as arguments. If the first one inherits from the second, it returns True. Else, it returns False.

>>> class A: pass

>>> class B(A): pass

>>> issubclass(B,A)

True

>>> issubclass(A,B)

False

Are methods and constructors the same thing?

No, there are subtle but considerable differences-

We must name a constructor __init()__; a method name can be anything.

Whenever we create an object, it executes a constructor; whenever we call a method, it executes a method.For one object, a constructor executes only once; a method can execute any number of times for one object.We use constructors to define and initialize non-static variables; we use methods to represent business logic to perform operations.

What is a Python module?

A module is a script in Python that defines import statements, functions, classes, and variables. It also holds runnable Python code. ZIP files and DLL files can be modules too. The module holds its name as a string that is in a global variable.

What are the file-related modules we have in Python?

We have the following libraries and modules that let us manipulate text and binary files on our file systems-
os
os.path
shutil

Explain, in brief, the uses of the modules sqlite3, ctypes, pickle, traceback, and itertools.

sqlite3- Helps with handling databases of type SQLite

ctypes- Lets create and manipulate C data types in Python

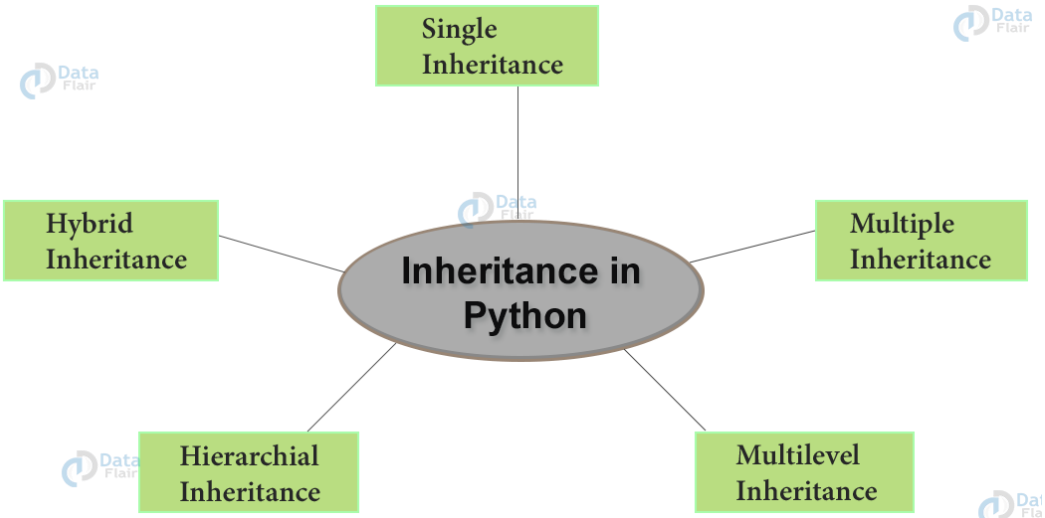
pickle- Lets put any data structure to external files

traceback- Allows extraction, formatting, and printing of stack traces

itertools– Supports working with permutations, combinations, and other useful iterables.

Explain inheritance in Python.

When one class inherits from another, it is said to be the child/derived/sub class inheriting from the parent/base/super class. It inherits/gains all members (attributes and methods).



Inheritance lets us reuse our code, and also makes it easier to create and maintain applications. Python supports the following kinds of inheritance:

- Single Inheritance- A class inherits from a single base class.
- Multiple Inheritance- A class inherits from multiple base classes.
- Multilevel Inheritance- A class inherits from a base class, which, in turn, inherits from another base class.
- Hierarchical Inheritance- Multiple classes inherit from a single base class.
- Hybrid Inheritance- Hybrid inheritance is a combination of two or more types of inheritance.

Explain memory management in Python.

Objects and data structures in Python lie on a private heap. The Python memory manager internally manages this. It delegates some work to object-specific allocators while ensuring they operate only within the private heap. Actually, the interpreter manages this heap; the user has no control over it- not even if they manipulate object pointers to memory blocks inside the heap. The Python memory manager allocates heap space to objects and other internal buffers on demand.

Write Python logic to count the number of capital letters in a file.

```
>>> import os

>>> os.chdir('C:\\Users\\lifei\\Desktop')

>>> with open('Today.txt') as today:

    count=0

    for i in today.read():

        if i.isupper():

            count+=1

    print(count)
```

OUTPUT:

26

How would you make a Python script executable on Unix?

For this to happen, two conditions must be met:

The script file's mode must be executable

The first line must begin with a hash(#). An example of this will be: #!/usr/local/bin/python

What functions or methods will you use to delete a file in Python?

For this, we may use remove() or unlink().

```
>>> import os
>>> os.chdir('C:\\Users\\lifei\\Desktop')
>>> os.remove('try.py')
>>>
```

When we go and check our Desktop, the file is gone. Let's go make it again so we can delete it again using unlink().

```
>>> os.unlink('try.py')
>>>
```

Both functions are the same, but unlink is the traditional Unix name for it.

Can you write a function to generate the following pyramid?

```
    *
   ***
  *****
 *******
*********
```

```
def pyramid(n):
  for row in range(n):
    for space in range(n-row):
      print(' ',end='')
    for star in range(row):
      print('*',end='')
    for star in range(row+1):
```

```
    print('*',end='')
  print()
pyramid(5)
```

How will you print the contents of a file?

```
>>> try:
    with open('tabs.txt','r') as f:
        print(f.read())
except IOError:
    print("File not found")
```

Explain lambda expressions. When would you use one?

When we want a function with a single expression, we can define it anonymously. A lambda expression may take input and returns a value.

To define the above function as a lambda expression, we type the following code in the interpreter:

```
>>> (lambda a,b:a if a>b else b)(3,3.5)
3.5
```

Here, a and b are the inputs. a if a>b else b is the expression to return. The arguments are 3 and 3.5.

It is possible to not have any inputs here.

```
>>> (lambda :print("Hi"))()
Hi
```

What is a generator?

Python generator produces a sequence of values to iterate on. This way, it is kind of an iterable.

We define a function that 'yields' values one by one, and then use a for loop to iterate on it.

```
>>> def squares(n):
i=1
while(i<=n):
yield i**2
i+=1
>>> for i in squares(7):
```

```
print(i)
```

```
1
```

```
4
```

```
9
```

```
16
```

```
25
```

```
36
```

```
49
```

So, what is an iterator, then?

An iterator returns one object at a time to iterate on. To create an iterator, we use the iter() function.

```
odds=iter([1,3,5,7,9])
```

Then, we call the next() function on it every time we want an object.

```
>>> next(odds)
1
>>> next(odds)
3
>>> next(odds)
5
>>> next(odds)
7
>>> next(odds)
9
```

And now, when we call it again, it raises a StopIteration exception. This is because it has reached the end of the values to iterate on.

```
>>> next(odds)
```

Diference between iterator and generator ?

They do, but there are subtle differences:
For a generator, we create a function. For an iterator, we use in-built functions iter() and next().
For a generator, we use the keyword ‘yield’ to yield/return an object at a time.
A generator may have as many ‘yield’ statements as you want.
A generator will save the states of the local variables every time ‘yield’ will pause the loop. An iterator does not use local variables; it only needs an iterable to iterate on.
Using a class, you can implement your own iterator, but not a generator.
Generators are fast, compact, and simpler.Iterators are more memory-efficient.

What is a decorator?

A decorator is a function that adds functionality to another function without modifying it. It wraps another function to add functionality to it. Take an example.

```
>>> def decor(func):
  def wrap():
      print("$$$$$$$$$$$$$$$$")
      func()
      print("$$$$$$$$$$$$$$$$")
return wrap
>>> @decor
def sayhi():
  print("Hi")
>>> sayhi()
$$$$$$$$$$$$$$$$
Hi
$$$$$$$$$$$$$$$$
```

Decorators are an example of metaprogramming, where one part of the code tries to change another.

What is Monkey Patching?

Monkey patching refers to modifying a class or module at runtime (dynamic modification). It extends Python code at runtime.

For example:

```
from pkg.module import MyClass
def sayhello(self):
      print("Hello")
MyClass.sayhello=sayhello
```

What do you mean by *args and **kwargs?

In cases when we don't know how many arguments will be passed to a function, like when we want to pass a list or a tuple of values, we use *args.

```
>>> def func(*args):
  for i in args:
     print(i)
>>> func(3,2,1,4,7)
3
2
1
4
7
```

**kwargs takes keyword arguments when we don't know how many there will be.

```
>>> def func(**kwargs):
  for i in kwargs:
     print(i,kwargs[i])
>>> func(a=1,b=2,c=7)
a.1
b.2
c.7
```

The words args and kwargs are a convention, and we can use anything in their place.

What is a closure in Python?

A closure in Python is said to occur when a nested function references a value in its enclosing scope. The whole point here is that it remembers the value.

```
>>> def A(x):
  def B():
     print(x)
  return B
>>> A(7)()
7
```

Are these statements optimal? If not, optimize them.

word=’word’

print(word.__len__())

No, these are not optimal. Let’s check the manual for this.

>>> help(str.__len__)

Help on wrapper_descriptor:

__len__(self, /)

Return len(self).

__len__ is a wrapper descriptor which in turn makes a call to len(). So why not skip the work and do just that instead?

The optimal solution:

>>> word='word'

>>> len(word)

4

What is the iterator protocol?

The iterator protocol for Python declares that we must make use of two functions to build an iterator- iter() and next().

iter()- To create an iterator

next()- To iterate to the next element

>>> a=iter([2,4,6,8,10])

>>> next(a)

2

What is tuple unpacking?

Suppose we have a tuple nums=(1,2,3). We can unpack its values into the variables a, b, and c. Here’s how:

>>> nums=(1,2,3)
>>> a,b,c=nums
>>> a
1

>>> b
2

```
>>> c
3
```

What will the following code output?

```
>>> a=1
>>> a,b=a+1,a+1
>>> a,b
```

The output is (2, 2). This code increments the value of a by 1 and assigns it to both a and b.
This is because this is a simultaneous declaration. The following code gives us the same:

```
>>> a=1
>>> b,a=a+1,a+1
>>> a,b
(2, 2)
```

What is a frozen set in Python?

First, let's discuss what a set is. A set is a collection of items, where there cannot be any duplicates. A set is also unordered.

```
>>> myset={1,3,2,2}
>>> myset

{1, 2, 3}
```

This means that we cannot index it.

```
>>> myset[0]

Traceback (most recent call last):

File "<pyshell#197>", line 1, in <module>

myset[0]

TypeError: 'set' object does not support indexing
```

However, a set is mutable. A frozen set is immutable. This means we cannot change its values. This also makes it eligible to be used as a key for a dictionary.

```
>>> myset=frozenset([1,3,2,2])

>>> myset

frozenset({1, 2, 3})
```

```
>>> type(myset)
<class 'frozenset'>
```

When you exit Python, is all memory deallocated?

Exiting Python deallocates everything except:

modules with circular references
Objects referenced from global namespaces
Parts of memory reserved by the C library

What is the Dogpile effect?

In case the cache expires, what happens when a client hits a website with multiple requests is what we call the dogpile effect.
To avoid this, we can use a semaphore lock. When the value expires, the first process acquires the lock and then starts to generate the new value.

Explain garbage collection with Python.

The following points are worth nothing for the garbage collector with CPython-
Python maintains a count of how many references there are to each object in memory
When a reference count drops to zero, it means the object is dead and Python can free the memory it allocated to that objectThe garbage collector looks for reference cycles and cleans them up
Python uses heuristics to speed up garbage collection. Recently created objects might as well be dead
The garbage collector assigns generations to each object as it is created . It deals with the younger generations first.

How will you use Python to read a random line from a file?

We can borrow the choice() method from the random module for this.

```
>>> import random
>>> lines=open('tabs.txt').read().splitlines()
>>> random.choice(lines)
'https://data-flair.training/blogs/category/python/'
```

Let's restart the IDLE and do this again.

```
>>> import random
>>> lines=open('tabs.txt').read().splitlines()
>>> random.choice(lines)
'https://data-flair.training/blogs/'
```

```
>>> random.choice(lines)
'https://data-flair.training/blogs/category/python/'

>>> random.choice(lines)
'https://data-flair.training/blogs/category/python/'

>>> random.choice(lines)
'https://data-flair.training/blogs/category/python/'
```

What is JSON? Describe in brief how you'd convert JSON data into Python data?

JSON stands for JavaScript Object Notation. It is a highly popular data format, and it stores data into NoSQL databases.
JSON is generally built on the following two structures:

A collection of <name,value> pairs
An ordered list of values.
Python supports JSON parsers. In fact, JSON-based data is internally represented as a dictionary in Python.
To convert JSON data into Python data, we use the load() function from the JSON module.

Differentiate between split(), sub(), and subn() methods of the remodule.

The re module is what we have for processing regular expressions with Python. Let's talk about the three methods we mentioned-
split()- This makes use of a regex pattern to split a string into a list
sub()- This looks for all substrings where the regex pattern matches, and replaces them with a different string
subn()- Like sub(), this returns the new string and the number of replacements made

How would you display a file's contents in reversed order?

Let's first get to the Desktop. We use the chdir() function/method form the os module for this.

```
>>> import os

>>> os.chdir('C:\\Users\\lifei\\Desktop')
```

The file we'll use for this is Today.txt, and it has the following contents:

```
OS, DBMS, DS, ADA

HTML, CSS, jQuery, JavaScript

Python, C++, Java
```

This sem's subjects

Debugger

itertools

container

Let's read the contents into a list, and then call reversed() on it:

```
>>> for line in reversed(list(open('Today.txt'))):
  print(line.rstrip())
```

container

itertools

Debugger

This sem's subjects

Python, C++, Java

HTML, CSS, jQuery, JavaScript

OS, DBMS, DS, ADA

Without the rstrip(), we would get blank lines between the output.

Can I dynamically load a module in Python?

Dynamic loading is where we do not load a module till we need it. This is slow, but lets us utilize the memory more efficiently. In Python, you can use the importlib module for this:

```
import importlib

module = importlib.import_module('my_package.my_module')]
```

How will you locally save an image using its URL address?

For this, we use the urllib module.

```
>>> import urllib.request

>>> urllib.request.urlretrieve('https://yt3.ggpht.com/a-/ACSszfE2YYTfvXCIVk4NjJdDfFSkSVrLBlalZwYsoA=s900-mo-c-c0xffffffff-rj-k-no','dataflair.jpg')

('dataflair.jpg', <http.client.HTTPMessage object at 0x02E90770>)
```

Optionally, what statements can you put under a try-except block?

We have two of those:

else- To run a piece of code when the try-block doesn't create an exception.

finally- To execute some piece of code regardless of whether there is an exception.

```
>>> try:
print("Hello")
except:
print("Sorry")
else:
print("Oh then")
finally:
print("Bye")
```

```
Hello
Oh then
Bye
```

How will you share global variables across modules?

To do this for modules within a single program, we create a special module, then import the config module in all modules of our application.

This lets the module be global to all modules.

List some pdb commands.

Some pdb commands include-

<b> — Add breakpoint

<c> — Resume execution

<s> — Debug step by step

<n> — Move to next line

<l> — List source code

<p> — Print an expression

What command do we use to debug a Python program?

To start debugging, we first open the command prompt and get to the location the file is at.

Microsoft Windows [Version 10.0.16299.248]

(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\lifei> cd Desktop

C:\Users\lifei\Desktop>

Then, we run the following command (for file try.py):

C:\Users\lifei\Desktop>python -m pdb try.py

> c:\users\lifei\desktop\try.py(1)<module>()

-> for i in range(5):

(Pdb)

Then, we can start debugging.

What is Tkinter?

Tkinter is a famous Python library with which you can craft a GUI. It provides support for different GUI tools and widgets like buttons, labels, text boxes,radio buttons, and more. These tools and widgets have attributes like dimensions, colors, fonts, colors, and more.

You can also import the tkinter module.

>>> import tkinter

>>> top=tkinter.Tk()

This will create a new window for you:

This creates a window with the title ‘My Game’. You can position your widgets on this.

Python Data Science Interview Questions

What is the process to calculate percentiles with NumPy?

Refer to the code below.

>>> import numpy as np

>>> arr=np.array([1,2,3,4,5])

>>> p=np.percentile(arr,50)

>>> p

3.0

How would you create an empty NumPy array?

To create an empty array with NumPy, we have two options:

a. Option 1

```
>>> import numpy

>>> numpy.array([])

array([], dtype=float64)
```

b. Option 2

```
>>> numpy.empty(shape=(0,0))

array([], shape=(0, 0), dtype=float64)
```

How is NumPy different from SciPy?

We have so far seen them used together. But they have subtle differences:

SciPy encompasses most new features

NumPy does hold some linear algebra functions

SciPy holds more fully-featured versions of the linear algebra modules and other numerical algorithms

NumPy has compatibility as one of its essential goals; it attempts to retain all features supported by any of its predecessors

NumPy holds the array data type and some basic operations: indexing, sorting, reshaping, and more

Explain different ways to create an empty NumPy array in Python.

We'll talk about two methods to create NumPy array-

First method-

```
>>> import numpy

>>> numpy.array([])

array([], dtype=float64)
```

Second method-

```
>>> numpy.empty(shape=(0,0))

array([], shape=(0, 0), dtype=float64)
```

What is monkey patching?

Dynamically modifying a class or module at run-time.

```
>>> class A:
  def func(self):
     print("Hi")
>>> def monkey(self):
     print "Hi, monkey"
>>> m.A.func = monkey
>>> a = m.A()
>>> a.func()
Hi, monkey
```

How will you find, in a string, the first word that rhymes with 'cake'?

For our purpose, we will use the function search(), and then use group() to get the output.

```
>>> import re
>>> rhyme=re.search('.ake','I would make a cake, but I hate to bake')
>>> rhyme.group()
'make'
```

And as we know, the function search() stops at the first match. Hence, we have our first rhyme to 'cake'.

Write a regular expression that will accept an email id. Use the re module.

```
>>> import re
>>> e=re.search(r'[0-9a-zA-Z.]+@[a-zA-Z]+\.(com|co\.in)$','abc@gmail.com')
>>> e.group()
'abc@gmail.com'
```

What is pickling and unpickling?

To create portable serialized representations of Python objects, we have the module 'pickle'.
It accepts a Python object (remember, everything in Python is an object).
It then converts it into a string representation and uses the dump() function to dump it into a file. We call this pickling. In contrast, retrieving objects from this stored string representation is termed 'unpickling'.

What is the MRO in Python?

MRO stands for Method Resolution Order. Talking of multiple inheritances, whenever we search for an attribute in a class, Python first searches in the current class. If found, its search is satiated. If not, it

moves to the parent class. It follows an approach that is left-to-right and depth-first. It goes Child, Mother, Father, Object.We can call this order a linearization of the class Child; the set of rules applied are the Method Resolution Order (MRO).

We can borrow the __mro__ attribute or the mro() method to get this.

How do we make forms in Python?

We use the cgi module for this; we borrow the FieldStorage class from it. It has the following attributes:

form.name: Name of field.
form.filename: Client-side filename for FTP transactions.
form.value: Value of field as string.
form.file: File object from which to read data.
form.type: Content type.
form.type_options: Options of 'content-type' line of HTTP request, returned as dictionary.
form.disposition: The field 'content-disposition'.form.disposition_options: Options for 'content-disposition'.form.headers: All HTTP headers returned as dictionary.

Python OOPS Interview Questions and Answers Is Python call-by-value or call-by-reference?
Python is neither call-by-value, nor call-by-reference. It is call-by-object-reference. Almost everything is an object in Python. Take this example:

```
>>> item='milk'
>>> groceries=[]
>>> groceries.append(item)
>>> groceries
['milk']

>>> items=groceries
>>> item='cheese'
>>> items.append(item)
>>> item
'cheese'

>>> groceries, items
(['milk', 'cheese'], ['milk', 'cheese'])
```

item is 'milk' and groceries is an empty list. We append item to the list of groceries; groceries is now the list ['milk']. Now, we assign groceries to the name items. item is now 'cheese'; let's append it to items. So now, the name item holds the value 'cheese', and items is the list ['milk', 'cheese']. But now, even groceries is the list ['milk', 'cheese']. The list groceries got updated too. With Python, you focus on objects, and not on names. It is one list object we modified for both names 'groceries' and 'items'.
Functions will modify values of mutable objects, but not immutable ones:

```
>>> a=1
```

```
>>> def up(num):
  num+=1
>>> up(a)
>>> a
1
------------------
------------------
>>> a=[1]
>>> def up(list):
  list[0]+=1
>>> up(a)
>>> a
[2]
```

Variables are names bound to an object, not aliases for actual memory locations.

Why do we need to overload operators?

To compare two objects, we can overload operators in Python. We understand 3>2. But what is orange>apple? Let's compare apples and oranges now.

```
>>> class fruit:
    def __init__(self,type,size):
        self.type='fruit'
        self.type=type
        self.size=size
    def __gt__(self,other):
        if self.size>other.size:
            return True
        return False
>>> orange=fruit('orange',7)
>>> apple=fruit('apple',8)
>>> apple>orange
True

>>> orange>apple
False
```

Why do we need the __init__() function in classes? What else helps?

__init__() is what we need to initialize a class when we initiate it. Let's take an example.

```
>>> class orange:
```

```
    def __init__(self):
        self.color='orange'
        self.type='citrus'
    def setsize(self,size):
        self.size=size
    def show(self):
        print(f"color: {self.color}, type: {self.type}, size: {self.size}")
>>> o=orange()
>>> o.setsize(7)
>>> o.show()
color: orange, type: citrus, size: 7
```

In this code, we see that it is necessary to pass the parameter 'self' to tell Python it has to work with this object.

Does Python support interfaces like Java does?

No. However, Abstract Base Classes (ABCs) are a feature from the abc module that let us declare what methods subclasses should implement.
Python supports this module since version 2.7.

What are accessors, mutators, and @property?

What we call getters and setters in languages like Java, we term accessors and mutators in Python.
In Java, if we have a user-defined class with a property 'x', we have methods like getX() and setX().
In Python, we have @property, which is syntactic sugar for property(). This lets us get and set variables without compromising on the conventions. For a detailed explanation on property, refer to Python property.
Consider multiple inheritances here. Suppose class C inherits from classes A and B as class C(A,B). Classes A and B both have their own versions of method func(). If we call func() from an object of class C, which version gets invoked?

In our article on Multiple Inheritance in Python, we discussed the Method Resolution Order (MRO).
C does not contain its own version of func(). Since the interpreter searches in a left-to-right fashion, it finds the method in A, and does not go to look for it in B.

What do you mean by overriding methods?

Suppose class B inherits from class A. Both have the method sayhello()- to each, their own version. B overrides the sayhello() of class A. So, when we create an object of class B, it calls the version that class B has.

```
>>> class A:
  def sayhello(self):
     print("Hello, I'm A")
>>> class B(A):
 def sayhello(self):
    print("Hello, I'm B")
>>> a=A()
>>> b=B()
>>> a.sayhello()
Hello, I’m A

>>> b.sayhello()
Hello, I’m B
```

How would you perform unit-testing on your Python code?

For this purpose, we have the module unittest in Python. It has the following members:

```
FunctionTestCase
SkipTest
TestCase
TestLoader
TestResult
TestSuite
TextTestResult
TextTestRunner
defaultTestLoader
expectedFailure
findTestCases
getTestCaseNames
installHandler
main
makeSuite
registerResult
removeHandler
removeResult
skip
skipIf
skipUnless
```

How is multithreading achieved in Python?

A thread is a lightweight process, and multithreading allows us to execute multiple threads at once. As you know, Python is a multithreaded language.

It has a multi-threading package.The GIL (Global Interpreter Lock) ensures that a single thread executes at a time. A thread holds the GIL and does a little work before passing it on to the next thread. This makes for an illusion of parallel execution.
But in reality, it is just threaded taking turns at the CPU. Of course, all the passing around adds overhead to the execution.

How is memory managed in Python?

Python has a private heap space to hold all objects and data structures. Being programmers, we cannot access it; it is the interpreter that manages it. But with the core API, we can access some tools. The Python memory manager controls the allocation.
Additionally, an inbuilt garbage collector recycles all unused memory so it can make it available to the heap space.

What is tuple unpacking?

First, let's discuss tuple packing. It is a way to pack a set of values into a tuple.

```
>>> mytuple=3,4,5
>>> mytuple
(3, 4, 5)
```

This packs 3, 4, and 5 into mytuple.
Now, we will unpack the values from the tuple into variables x, y, and z.

```
>>> x,y,z=mytuple
>>> x+y+z
12
```

What is a namedtuple?

A namedtuple will let us access a tuple's elements using a name/label. We use the function namedtuple() for this, and import it from collections.

```
>>> from collections import namedtuple
>>> result=namedtuple('result','Physics Chemistry Maths') #format
>>> Ayushi=result(Physics=86,Chemistry=95,Maths=86) #declaring the tuple
>>> Ayushi.Chemistry
95
```

As you can see, it let us access the marks in Chemistry using the Chemistry attribute of object Ayushi.

How do you create your own package in Python?

We know that a package may contain sub-packages and modules. A module is nothing but Python code. To create a Python package of our own, we create a directory and create a file __init__.py in it. We leave it empty. Then, in that package, we create a module(s) with whatever code we want.

Have you heard of the yield keyword in Python?

Yes, I have. This keyword bears the ability to turn any function into a generator. Much like the standard return keyword, but returns a generatorobject.
It is also true that one function may observe multiple yields.

```
>>> def odds(n):
    odd=[i for i in range(n+1) if i%2!=0]
    for i in odd:
        yield i
>>> for i in odds(8):
    print(i)
1
3
5
7
```

If a function does not have a return statement, is it valid?

Very conveniently. A function that doesn't return anything returns a None object.
Not necessarily does the return keyword mark the end of a function; it merely ends it when present in the function. Normally, a block of code marks a function and where it ends, the function body ends.

So, this was the last category of our Python programming interview questions and answers.
Hope this helped you. If you have more python interview questions for experienced or freshers or any interview experience do share with us through comments.

Python example Program:

Example: Kilometers to Miles

```
# Taking kilometers input from the user
kilometers = float(input("Enter value in kilometers: "))

# conversion factor
conv_fac = 0.621371

# calculate miles
```

```
miles = kilometers * conv_fac
print('%0.2f kilometers is equal to %0.2f miles' %(kilometers,miles))
```

Program to check if a number is prime or not

```
num = 407

# To take input from the user
#num = int(input("Enter a number: "))

# prime numbers are greater than 1
if num > 1:
  # check for factors
  for i in range(2,num):
     if (num % i) == 0:
        print(num,"is not a prime number")
        print(i,"times",num//i,"is",num)
        break
  else:
     print(num,"is a prime number")

# if input number is less than
# or equal to 1, it is not prime
else:
  print(num,"is not a prime number")
```

Python Program to Check if a Number is Positive, Negative or 0

```
  num = float(input("Enter a number: "))
if num > 0:
  print("Positive number")
elif num == 0:
  print("Zero")
else:
  print("Negative number")
```

Python Program to Check if a Number is Odd or Even

```
# Python program to check if the input number is odd or even.
# A number is even if division by 2 gives a remainder of 0.
# If the remainder is 1, it is an odd number.
```

```
num = int(input("Enter a number: "))
if (num % 2) == 0:
  print("{0} is Even".format(num))
else:
  print("{0} is Odd".format(num))
```

Python Program to Check Leap Year

A leap year is exactly divisible by 4 except for century years (years ending with 00). The century year is a leap year only if it is perfectly divisible by 400. For example,

2017 is not a leap year
1900 is a not leap year
2012 is a leap year
2000 is a leap year

```
# Python program to check if year is a leap year or not

year = 2000

# To get year (integer input) from the user
# year = int(input("Enter a year: "))

if (year % 4) == 0:
  if (year % 100) == 0:
     if (year % 400) == 0:
        print("{0} is a leap year".format(year))
     else:
        print("{0} is not a leap year".format(year))
  else:
     print("{0} is a leap year".format(year))
else:
  print("{0} is not a leap year".format(year))
```

Python Program to Find the Largest Among Three Numbers

```
# Python program to find the largest number among the three input numbers

# change the values of num1, num2 and num3
# for a different result
num1 = 10
```

```
num2 = 14
num3 = 12

# uncomment following lines to take three numbers from user
#num1 = float(input("Enter first number: "))
#num2 = float(input("Enter second number: "))
#num3 = float(input("Enter third number: "))

if (num1 >= num2) and (num1 >= num3):
  largest = num1
elif (num2 >= num1) and (num2 >= num3):
  largest = num2
else:
  largest = num3

print("The largest number is", largest)
```

Python Program to Print all Prime Numbers in an Interval

```
# Python program to display all the prime numbers within an interval

lower = 900
upper = 1000

print("Prime numbers between", lower, "and", upper, "are:")

for num in range(lower, upper + 1):
  # all prime numbers are greater than 1
  if num > 1:
     for i in range(2, num):
        if (num % i) == 0:
           break
     else:
        print(num)
```

Python Program to Find the Factorial of a Number

```
# Python program to find the factorial of a number provided by the user.

# change the value for a different result
num = 7

# To take input from the user
```

```
#num = int(input("Enter a number: "))

factorial = 1

# check if the number is negative, positive or zero
if num < 0:
   print("Sorry, factorial does not exist for negative numbers")
elif num == 0:
   print("The factorial of 0 is 1")
else:
   for i in range(1,num + 1):
       factorial = factorial*i
   print("The factorial of",num,"is",factorial)
```

Python Program to Print the Fibonacci sequence

```
# Program to display the Fibonacci sequence up to n-th term

nterms = int(input("How many terms? "))

# first two terms
n1, n2 = 0, 1
count = 0

# check if the number of terms is valid
if nterms <= 0:
   print("Please enter a positive integer")
elif nterms == 1:
   print("Fibonacci sequence upto",nterms,":")
   print(n1)
else:
   print("Fibonacci sequence:")
   while count < nterms:
       print(n1)
       nth = n1 + n2
       # update values
       n1 = n2
       n2 = nth
       count += 1
```

Python Program to Check Armstrong Number

```
# Python program to check if the number is an Armstrong number or not
```

```
# take input from the user
num = int(input("Enter a number: "))

# initialize sum
sum = 0

# find the sum of the cube of each digit
temp = num
while temp > 0:
  digit = temp % 10
  sum += digit ** 3
  temp //= 10

# display the result
if num == sum:
  print(num,"is an Armstrong number")
else:
  print(num,"is not an Armstrong number")
```

Output 1

```
Enter a number: 663
663 is not an Armstrong number
```

Output 2

```
Enter a number: 407
407 is an Armstrong number
```

Source Code: Check Armstrong number of n digits

```
num = 1634

# Changed num variable to string,
# and calculated the length (number of digits)
order = len(str(num))

# initialize sum
sum = 0

# find the sum of the cube of each digit
temp = num
while temp > 0:
  digit = temp % 10
```

```
  sum += digit ** order
  temp //= 10

# display the result
if num == sum:
  print(num,"is an Armstrong number")
else:
  print(num,"is not an Armstrong number")
```

Python Program to Find the Sum of Natural Numbers

```
# Sum of natural numbers up to num

num = 16

if num < 0:
  print("Enter a positive number")
else:
  sum = 0
  # use while loop to iterate until zero
  while(num > 0):
     sum += num
     num -= 1
  print("The sum is", sum)
```

Python Program To Display Powers of 2 Using Anonymous Function

```
# Display the powers of 2 using anonymous function

terms = 10

# Uncomment code below to take input from the user
# terms = int(input("How many terms? "))

# use anonymous function
result = list(map(lambda x: 2 ** x, range(terms)))

print("The total terms are:",terms)
for i in range(terms):
  print("2 raised to power",i,"is",result[i])
```

Python Program to Find Numbers Divisible by Another Number

```
# Take a list of numbers
my_list = [12, 65, 54, 39, 102, 339, 221,]
```

```
# use anonymous function to filter
result = list(filter(lambda x: (x % 13 == 0), my_list))

# display the result
print("Numbers divisible by 13 are",result)
```

Python Program to Convert Decimal to Binary, Octal and Hexadecimal

```
# Python program to convert decimal into other number systems
dec = 344

print("The decimal value of", dec, "is:")
print(bin(dec), "in binary.")
print(oct(dec), "in octal.")
print(hex(dec), "in hexadecimal.")
```

Python Program to Find ASCII Value of Character

```
# Program to find the ASCII value of the given character

c = 'p'
print("The ASCII value of '" + c + "' is", ord(c))
```

Python Program to Make a Simple Calculator

```
# Program make a simple calculator

# This function adds two numbers
def add(x, y):
   return x + y

# This function subtracts two numbers
def subtract(x, y):
   return x - y

# This function multiplies two numbers
def multiply(x, y):
   return x * y

# This function divides two numbers
def divide(x, y):
   return x / y


print("Select operation.")
print("1.Add")
```

```
print("2.Subtract")
print("3.Multiply")
print("4.Divide")

while True:
  # Take input from the user
  choice = input("Enter choice(1/2/3/4): ")

  # Check if choice is one of the four options
  if choice in ('1', '2', '3', '4'):
    num1 = float(input("Enter first number: "))
    num2 = float(input("Enter second number: "))

    if choice == '1':
      print(num1, "+", num2, "=", add(num1, num2))

    elif choice == '2':
      print(num1, "-", num2, "=", subtract(num1, num2))

    elif choice == '3':
      print(num1, "*", num2, "=", multiply(num1, num2))

    elif choice == '4':
      print(num1, "/", num2, "=", divide(num1, num2))
    break
  else:
    print("Invalid Input")
```

Python Program to Shuffle Deck of Cards

```
# Python program to shuffle a deck of card

# importing modules
import itertools, random

# make a deck of cards
deck = list(itertools.product(range(1,14),['Spade','Heart','Diamond','Club']))

# shuffle the cards
random.shuffle(deck)

# draw five cards
print("You got:")
```

```
for i in range(5):
  print(deck[i][0], "of", deck[i][1])
```

Python Program to Display Calendar

```
# Program to display calendar of the given month and year

# importing calendar module
import calendar

yy = 2014  # year
mm = 11    # month

# To take month and year input from the user
# yy = int(input("Enter year: "))
# mm = int(input("Enter month: "))

# display the calendar
print(calendar.month(yy, mm))
```

Python Program to Find Sum of Natural Numbers Using Recursion

```
# Python program to find the sum of natural using recursive function

def recur_sum(n):
  if n <= 1:
     return n
  else:
     return n + recur_sum(n-1)

# change this value for a different result
num = 16

if num < 0:
  print("Enter a positive number")
else:
  print("The sum is",recur_sum(num))
```

Python Program to Find Factorial of Number Using Recursion

```
# Factorial of a number using recursion

def recur_factorial(n):
  if n == 1:
```

```
    return n
  else:
    return n*recur_factorial(n-1)

num = 7

# check if the number is negative
if num < 0:
  print("Sorry, factorial does not exist for negative numbers")
elif num == 0:
  print("The factorial of 0 is 1")
else:
  print("The factorial of", num, "is", recur_factorial(num))
```

Python Program to Transpose a Matrix

```
# Program to transpose a matrix using a nested loop

X = [[12,7],
   [4 ,5],
   [3 ,8]]

result = [[0,0,0],
       [0,0,0]]

# iterate through rows
for i in range(len(X)):
  # iterate through columns
  for j in range(len(X[0])):
    result[j][i] = X[i][j]

for r in result:
  print(r)
```

Python Program to Add Two Matrices

```
# Program to add two matrices using nested loop

X = [[12,7,3],
   [4 ,5,6],
   [7 ,8,9]]

Y = [[5,8,1],
   [6,7,3],
```

```
   [4,5,9]]

result = [[0,0,0],
        [0,0,0],
        [0,0,0]]

# iterate through rows
for i in range(len(X)):
  # iterate through columns
  for j in range(len(X[0])):
      result[i][j] = X[i][j] + Y[i][j]

for r in result:
  print(r)
```

Python Program to Convert Decimal to Binary Using Recursion

```
# Function to print binary number using recursion
def convertToBinary(n):
  if n > 1:
      convertToBinary(n//2)
  print(n % 2,end = '')

# decimal number
dec = 34

convertToBinary(dec)
print()
```

Python Program to Multiply Two Matrices

```
# Program to multiply two matrices using nested loops

# 3x3 matrix
X = [[12,7,3],
   [4 ,5,6],
   [7 ,8,9]]
# 3x4 matrix
Y = [[5,8,1,2],
   [6,7,3,0],
   [4,5,9,1]]
# result is 3x4
result = [[0,0,0,0],
        [0,0,0,0],
```

```
    [0,0,0,0]]

# iterate through rows of X
for i in range(len(X)):
  # iterate through columns of Y
  for j in range(len(Y[0])):
    # iterate through rows of Y
    for k in range(len(Y)):
      result[i][j] += X[i][k] * Y[k][j]

for r in result:
  print(r)
```

Python Program to Check Whether a String is Palindrome or Not

```
# Program to check if a string is palindrome or not

my_str = 'aIbohPhoBiA'

# make it suitable for caseless comparison
my_str = my_str.casefold()

# reverse the string
rev_str = reversed(my_str)

# check if the string is equal to its reverse
if list(my_str) == list(rev_str):
  print("The string is a palindrome.")
else:
  print("The string is not a palindrome.")
```

Python Program to Remove Punctuations From a String

```
# define punctuation
punctuations = '''!()-[]{};:'"\,<>./?@#$%^&*_~'''

my_str = "Hello!!!, he said ---and went."

# To take input from the user
# my_str = input("Enter a string: ")

# remove punctuation from the string
no_punct = ""
for char in my_str:
  if char not in punctuations:
    no_punct = no_punct + char
```

```
# display the unpunctuated string
print(no_punct)
```

Python Program to Sort Words in Alphabetic Order

```
# Program to sort alphabetically the words form a string provided by the user

my_str = "Hello this Is an Example With cased letters"

# To take input from the user
#my_str = input("Enter a string: ")

# breakdown the string into a list of words
words = my_str.split()

# sort the list
words.sort()

# display the sorted words

print("The sorted words are:")
for word in words:
  print(word)
```

Python Program to Merge Mails

```
# Python program to mail merger
# Names are in the file names.txt
# Body of the mail is in body.txt

# open names.txt for reading
with open("names.txt",'r',encoding = 'utf-8') as names_file:

  # open body.txt for reading
  with open("body.txt",'r',encoding = 'utf-8') as body_file:

    # read entire content of the body
    body = body_file.read()

    # iterate over names
    for name in names_file:
      mail = "Hello "+name+body

      # write the mails to individual files
      with open(name.strip()+".txt",'w',encoding = 'utf-8') as mail_file:
```

```
        mail_file.write(mail)
```

For this program, we have written all the names in separate lines in the file "names.txt". The body is in the "body.txt" file.
We open both the files in reading mode and iterate over each name using a for loop. A new file with the name "[name].txt" is created,
where name is the name of that person.

We use strip() method to clean up leading and trailing whitespaces (reading a line from the file also reads the newline '\n' character).
Finally, we write the content of the mail into this file using the write() method.

Python Program to Count the Number of Each Vowel

```
# Program to count the number of each vowels

# string of vowels
vowels = 'aeiou'

ip_str = 'Hello, have you tried our tutorial section yet?'

# make it suitable for caseless comparisions
ip_str = ip_str.casefold()

# make a dictionary with each vowel a key and value 0
count = {}.fromkeys(vowels,0)

# count the vowels
for char in ip_str:
  if char in count:
     count[char] += 1

print(count)
```

Python Program to Illustrate Different Set Operations

```
# Program to perform different set operations like in mathematics

# define three sets
E = {0, 2, 4, 6, 8};
N = {1, 2, 3, 4, 5};

# set union
print("Union of E and N is",E | N)
```

```
# set intersection
print("Intersection of E and N is",E & N)

# set difference
print("Difference of E and N is",E - N)

# set symmetric difference
print("Symmetric difference of E and N is",E ^ N)
```

Python Program to Find Hash of File

```
# Python rogram to find the SHA-1 message digest of a file

# importing the hashlib module
import hashlib

def hash_file(filename):
  """This function returns the SHA-1 hash
  of the file passed into it"""

  # make a hash object
  h = hashlib.sha1()

  # open file for reading in binary mode
  with open(filename,'rb') as file:

      # loop till the end of the file
      chunk = 0
      while chunk != b'':
          # read only 1024 bytes at a time
          chunk = file.read(1024)
          h.update(chunk)

  # return the hex representation of digest
  return h.hexdigest()

message = hash_file("track1.mp3")
print(message)
```

Python Program to Find the Size (Resolution) of a Image

```
def jpeg_res(filename):
  """This function prints the resolution of the jpeg image file passed into it"""
```

```
  # open image for reading in binary mode
  with open(filename,'rb') as img_file:

      # height of image (in 2 bytes) is at 164th position
      img_file.seek(163)

      # read the 2 bytes
      a = img_file.read(2)

      # calculate height
      height = (a[0] << 8) + a[1]

      # next 2 bytes is width
      a = img_file.read(2)

      # calculate width
      width = (a[0] << 8) + a[1]

  print("The resolution of the image is",width,"x",height)

jpeg_res("img1.jpg")
```

What is NumPy?

NumPy is a python library used for working with arrays.
It also has functions for working in domain of linear algebra, fourier transform, and matrices.
NumPy was created in 2005 by Travis Oliphant. It is an open source project and you can use it freely.
NumPy stands for Numerical Python.

Why Use NumPy ?

In Python we have lists that serve the purpose of arrays, but they are slow to process.
NumPy aims to provide an array object that is up to 50x faster that traditional Python lists.
The array object in NumPy is called ndarray, it provides a lot of supporting functions that make working with ndarray very easy.
Arrays are very frequently used in data science, where speed and resources are very important.

Why is NumPy Faster Than Lists?

NumPy arrays are stored at one continuous place in memory unlike lists, so processes can access and manipulate them very efficiently.
This behavior is called locality of reference in computer science.
This is the main reason why NumPy is faster than lists. Also it is optimized to work with latest CPU architectures.

Installation of NumPy

C:\Users\Your Name>pip install numpy

```
import numpy
arr = numpy.array([1, 2, 3, 4, 5])
print(arr)
```

check the version

```
import numpy as np
print(np.__version__)
```

Create a NumPy ndarray Object

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
print(arr)
print(type(arr))
```

output:

```
[1 2 3 4 5]
<class 'numpy.ndarray'>
```

type(): This built-in Python function tells us the type of the object passed to it. Like in above code it shows that arr is numpy.ndarray type.

0-D Arrays

0-D arrays, or Scalars, are the elements in an array. Each value in an array is a 0-D array.

Example

Create a 0-D array with value 42

```
import numpy as np
arr = np.array(42)
print(arr)
```

Create a 1-D array containing the values 1,2,3,4,5:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
print(arr)
```

2-D Arrays

An array that has 1-D arrays as its elements is called a 2-D array.
These are often used to represent matrix or 2nd order tensors.
NumPy has a whole sub module dedicated towards matrix operations called numpy.mat

Create a 2-D array containing two arrays with the values 1,2,3 and 4,5,6:

```
import numpy as np
arr = np.array([[1, 2, 3], [4, 5, 6]])
print(arr)
```

Create a 3-D array with two 2-D arrays, both containing two arrays with the values 1,2,3 and 4,5,6:

```
import numpy as np
arr = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])
print(arr)
```

Check Number of Dimensions?

NumPy Arrays provides the ndim attribute that returns an integer that tells us how many dimensions the array have.

Example
Check how many dimensions the arrays have:

```
import numpy as np

a = np.array(42)
b = np.array([1, 2, 3, 4, 5])
c = np.array([[1, 2, 3], [4, 5, 6]])
d = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])

print(a.ndim)
print(b.ndim)
print(c.ndim)
print(d.ndim)
```

Higher Dimensional Arrays
An array can have any number of dimensions.
When the array is created, you can define the number of dimensions by using the ndmin argument.

Example
Create an array with 5 dimensions and verify that it has 5 dimensions:

```
import numpy as np
arr = np.array([1, 2, 3, 4], ndmin=5)
print(arr)
print('number of dimensions :', arr.ndim)
```

NumPy Array Indexing

```
import numpy as np
arr = np.array([1, 2, 3, 4])
print(arr[0])
```

Example

Access the 2nd element on 1st dim:

```
import numpy as np
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])
print('2nd element on 1st dim: ', arr[0, 1])
```

Access the 5th element on 2nd dim:

```
import numpy as np
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])
print('5th element on 2nd dim: ', arr[1, 4])
```

Example

Access the third element of the second array of the first array:

```
import numpy as np
arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
print(arr[0, 1, 2])
```

Example Explained

arr[0, 1, 2] prints the value 6.

And this is why:

The first number represents the first dimension, which contains two arrays:
[[1, 2, 3], [4, 5, 6]]
and:
[[7, 8, 9], [10, 11, 12]]
Since we selected 0, we are left with the first array:
[[1, 2, 3], [4, 5, 6]]

The second number represents the second dimension, which also contains two arrays:
[1, 2, 3]
and:
[4, 5, 6]
Since we selected 1, we are left with the second array:
[4, 5, 6]

The third number represents the third dimension, which contains three values:
4
5
6
Since we selected 2, we end up with the third value:
6

Negative Indexing
Use negative indexing to access an array from the end.

Example
Print the last element from the 2nd dim:

```
import numpy as np
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])
print('Last element from 2nd dim: ', arr[1, -1])
```

output:
Last element from 2nd dim: 10

NumPy Array Slicing

Slicing in python means taking elements from one given index to another given index.
Slice elements from index 1 to index 5 from the following array:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[1:5])
```

STEP
Use the step value to determine the step of the slicing:

Example
Return every other element from index 1 to index 5:

```
import numpy as np
```

```
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[1:5:2])
```

output
[2 4]

From the second element, slice elements from index 1 to index 4 (not included):

```
import numpy as np
arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])
print(arr[1, 1:4])
```

Example
From both elements, slice index 1 to index 4 (not included), this will return a 2-D array:

```
import numpy as np
arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])
print(arr[0:2, 1:4])
```

Below is a list of all data types in NumPy and the characters used to represent them.

i - integer
b - boolean
u - unsigned integer
f - float
c - complex float
m - timedelta
M - datetime
O - object
S - string
U - unicode string
V - fixed chunk of memory for other type (void)

Get the data type of an array object:

```
import numpy as np

arr = np.array([1, 2, 3, 4])

print(arr.dtype)
```

output:
int64

Example
Get the data type of an array containing strings:

```
import numpy as np
arr = np.array(['apple', 'banana', 'cherry'])
print(arr.dtype)
```

<U6

NumPy Array Copy Vs View

The main difference between a copy and a view of an array is that the copy is a new array, and the view is just a view of the original array.
The copy owns the data and any changes made to the copy will not affect original array, and any changes made to the original array will not affect the copy.
The view does not own the data and any changes made to the view will affect the original array, and any changes made to the original array will affect the view.
Make a copy, change the original array, and display both arrays:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])
x = arr.copy()
arr[0] = 42

print(arr)
print(x)
```

output:
[42 2 3 4 5]
[1 2 3 4 5]

The copy SHOULD NOT be affected by the changes made to the original array.

VIEW:
Example
Make a view, change the original array, and display both arrays:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])
x = arr.view()
```

```
arr[0] = 42

print(arr)
print(x)
```

```
[42  2  3  4  5]
[42  2  3  4  5]
```

The view SHOULD be affected by the changes made to the original array.

Make Changes in the VIEW:
Example
Make a view, change the view, and display both arrays:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])
x = arr.view()
x[0] = 31
print(arr)
print(x)
```

```
[31  2  3  4  5]
[31  2  3  4  5]
```

The original array SHOULD be affected by the changes made to the view.