Queries:

Metaclasses in python.

Why is list compreshension faster?

Monkey Patching

1. What is @classmethod?
2. What is @staticmethod?
3. When to use @classmethod and @staticmethod?
4. Why are strings immutable in Python?
5. Is python interpreted or compiled language?
6. When to use a tuple and when to use a list?
7. What is monkey patching?
8. new() in python
9. How is an object constructed and destructed in Python?

10. How to debug in Python using PDP?

11.dir() method in Python?

12. Factory method in Python?

13. High level functions in Python?

14. Simple code to create a decorator?

15. How to create a static function in python?

16. Why is the pyc file created?

17. Kwargs vs args? Which comes first?

1. Is python interpreted or compiled?

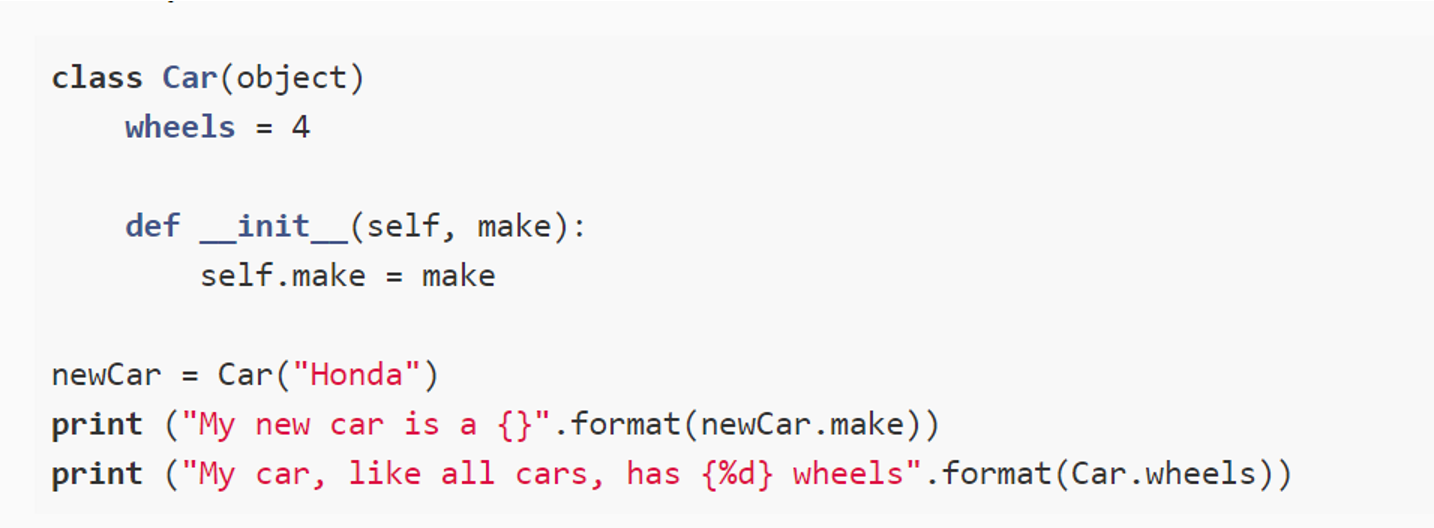
* A language is nerver interpreted or compiled language.
* **Python has an implementation of an interpreter that also has a built in complier**.
* If we create a abc.py file and import xyz.py file and run abc.py file, then only xyz.pyc file will be created as its imported and no abc.pyc file would be created.
* To create a pyc file we need to run – python –m py\_compile “fileName” or in code type:
* >>> import py\_compile
* >>> py\_compile.compile('abc.py')
* **A pyc file is a compiled file that is in machine language**.
* In python 3.2 and later, the compiled files are stored in \_\_pycache\_\_ folder.
* **This helps to increase the startup time for the application as when the program runs for the next time, these compiled files are used instead them creating a new compiled files.**
* So once a pyc file is created, next time when the program is executed, if the pyc is not older than the py file, python will execute the existing pyc file, making the loading faster by avoiding compilation.
* **Pyc file is created to load the file faster.**
* **Pyc file is only created for the imported modules and not for the main module.**

1. **Class Variables vs Instance Variables**:
2. Class Variables: declared without self keyword outside a function.

* Same value of the class variables is used for all the class instances.
* So this is like a static keyword.
* Can be accessed using the classname
* **Can be accessed using the class object as well.**

1. Instance Variables:

* They have different values for each object instance.
* Defined inside a function and prefixed with ‘self’.
* Can be accessed using object of the class.
* Can’t be accessed using the class name. Class name has no idea about the instance.



1. **\_ and \_\_ in python**.
2. Underscore before variables:

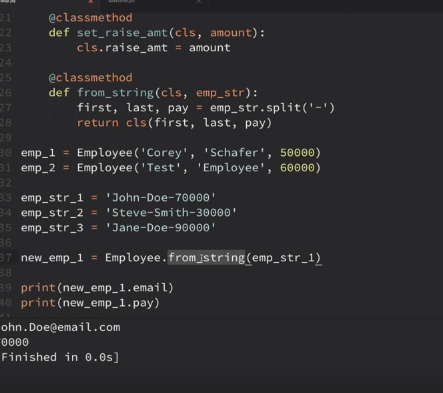
* Its kind of a convention to tell anyone (other programers or ourselves) that this name is to **treated as ‘private’ variable for internal purposes only.**
* **Also, when we do ‘from Module import \*’, the variables with \_ before them won’t be imported.**
* **\_ variable is not part of the public API of the application. Its just for internal use.**

1. Double underscore:

* Python mangles the names of these variable/methods with the class names.
* So \_\_variable becomes \_classname\_\_variable.
* **This is to avoid name clashes with names definded in the subclasses**.
* This also makes it difficult for the subclasses to override these methods, hence its similar to final keyword in java.
* **It’s not an enforced protection**.

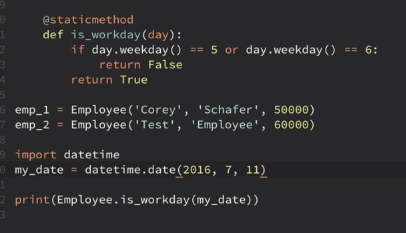
1. **Types of Methods in Python:**
2. Class Methods:

* These are the methods created with @classmethod decorator.
* They take the class itself as the first argument by default. Conventionally, this argument is named as ‘cls’.
* **Class method can modify the class state**
* **Class method can’t change object instance state.**
* Classmethods are basically used to provide multiple ways to create our objects.

****

1. **Staticmethods:**

* Methods created with @staticmethod decorator.
* Don’t use either the instance arg and the class arg.
* Mainly used when we don’t need the class or static reference inside the method.
* **Used when we want to make some change that does not affect the class and object state**
* Usually created for helper functions that should not update any object

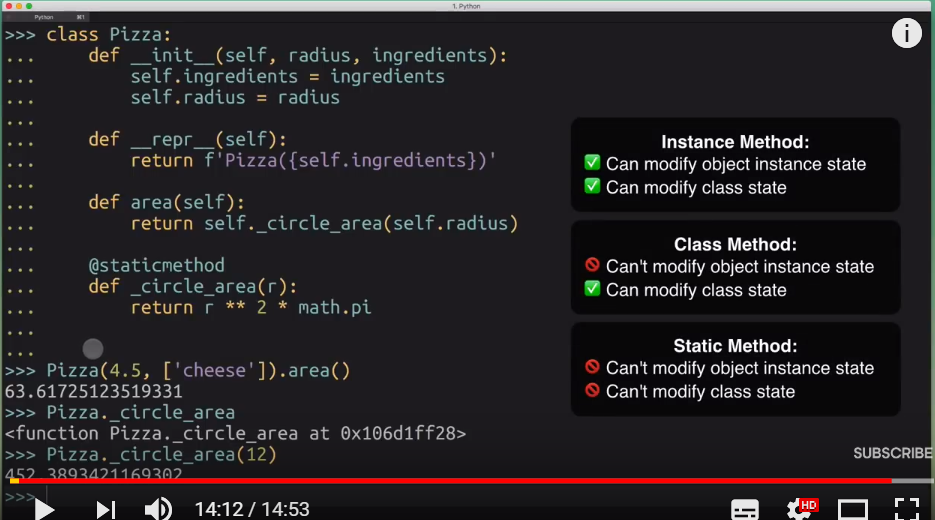
****

1. Instance Method:

* Its created without any decorator.
* It takes ‘self’ as the first argument by convention.
* **It can modify object instance state**
* **It can modify class state.**

**Note: Object of a class can call the instance, static and class methods.**

**ClassName can only call class method and static method. Can’t call instance method.**



1. **\*args and \*\*kwargs**:

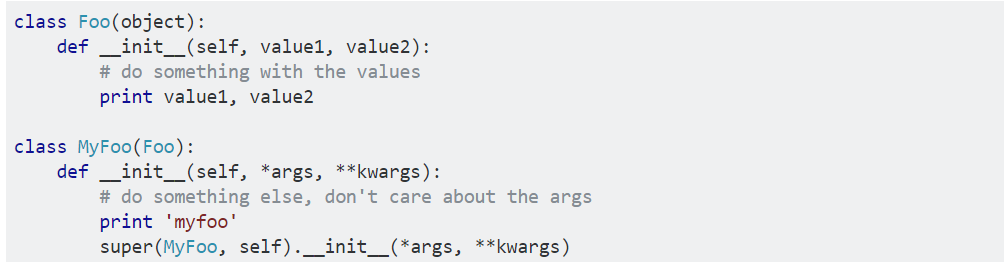
* These are optional arguments. We can use it to make function react different based on what kind of arguments we pass

1. \*args:

* is a tuple of positional arguments that are read based on their position .
* We use it when we are not sure how many arguments might be passed to your function.

1. \*\*kwargs: is a dictionary of named arguments that we have not defined in advance

* One example where these can be used is in the ‘derived class’. Here we can extend the behavior of the base class without having to know much about the base class.



The explicit (mandatory) arguments get the value first and then everything else is passed into \*args and \*\*kwargs

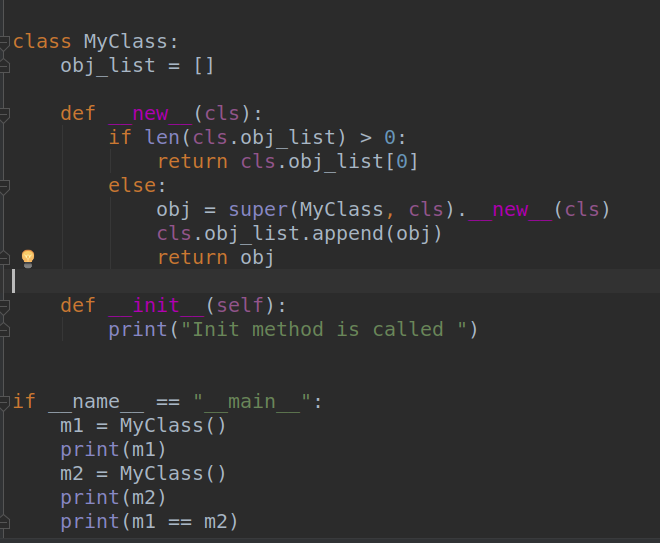
When used both, \*args must be used before \*\*kwargs

1. **\_\_new\_\_ and \_\_init\_\_ methods:**

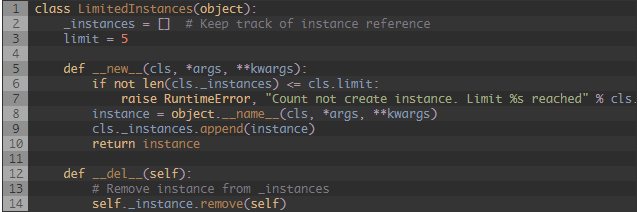
* Use \_\_new\_\_ when you need to control the creation of a new instance. Use \_\_init\_\_ when you need to control initialization of a new instance.
* \_\_new\_\_ is the first step of instance creation. It's called first, and is responsible for returning a new instance of your class. In contrast, \_\_init\_\_ doesn't return anything; it's only responsible for initializing the instance after it's been created.
* **\_\_init\_\_ won’t be called if \_\_new\_\_ does not return a new instance.**
* \_\_init\_\_ should never have a return type or we will get an error.
* In general, you shouldn't need to override \_\_new\_\_ unless you're subclassing an immutable type like str, int, unicode or tuple*.*
* We can create an new instance from a \_\_new\_\_ method by calling the parent classes \_\_new\_\_ method:

super(CurrentClassName, cls).\_\_new\_\_(cls, \*args, \*\*kwargs)

b) Creating Singleton pattern with \_\_new\_\_:



c) Controlling the number of instances to be created in python:



1. Property:

* Used to implement Getter Setter and Delete functionality on a class’s attributes.

1. Polymorphism:

* Means a thing can do different things based on a situation.
* For example, + operator

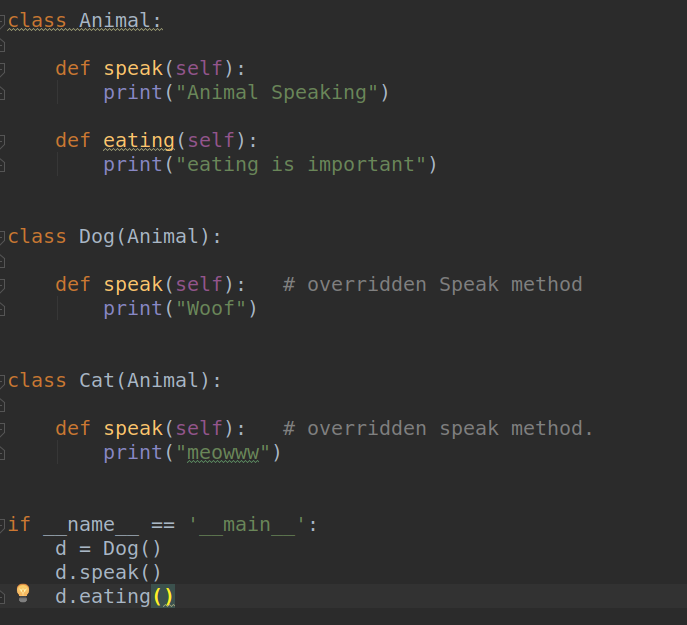
1 + 2 is 3

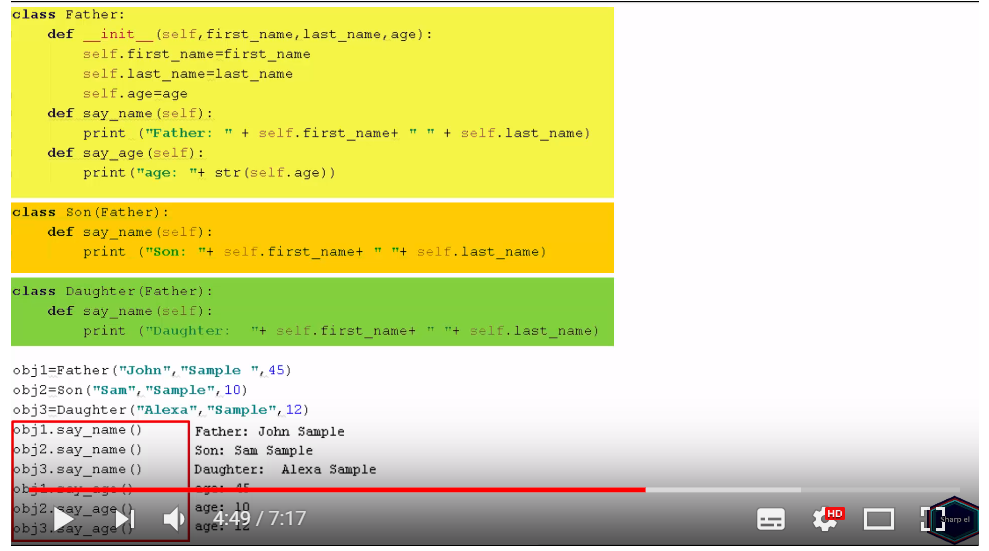
“he” + “llo” is hello

* With Class, an object can used to refer to the object of the same class and the subclass (overriding).

1. Method overloading vs Method Overriding:

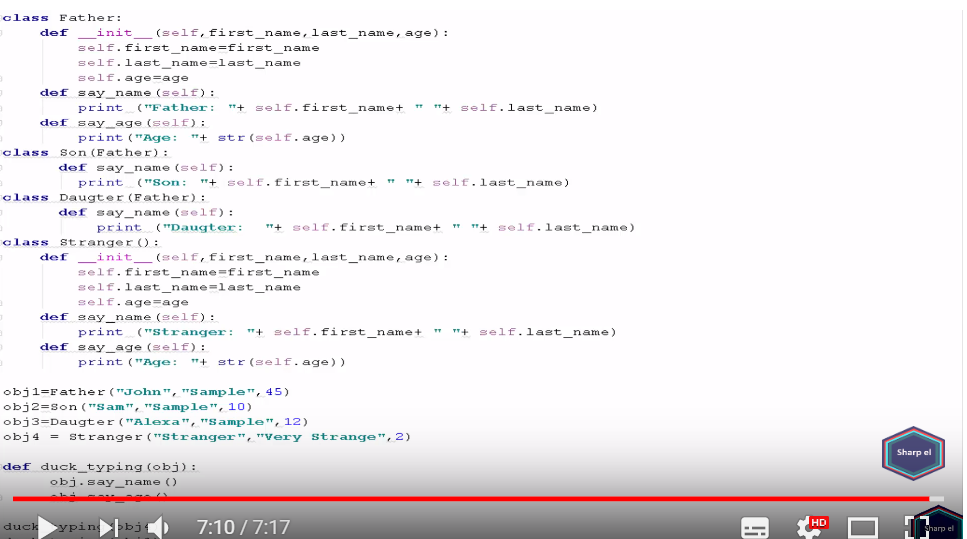
* Overriding: Derived class creates a function with same name as in the Parent class but with different functionality.
* Inheritance is required for Overriding.





* Overloading:
* Its not supported in Python.
* We can’t have more than one function with same name in the same class in Python.

1. Duck Typing:



* The idea is that we don’t need a type to invoke an existing method on an object. If a method is defined on it, we can invoke it.
* We don’t care what the object actual is.
* What's important in duck typing is what the object can actually do, rather than what the object is

Good read - <https://stackoverflow.com/a/40434829/4418897>

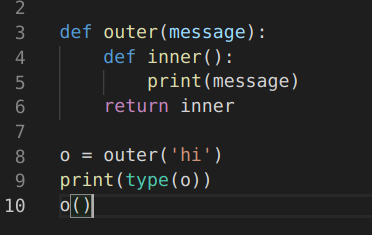
1. Higher order functions:
2. In Python, functions are treated as first class objects, allowing you to perform the following operations on functions.

- A function can take one or more functions as arguments

- A function can be returned as a result of another function

1. Clousers in Python:

a) A closure **is a way of keeping alive a variable even when the function has returned**.



1. So in simple terms, a clouser is an inner method, **That remembers and has access to variables to the local scope in which it was created, even after the outer function has finished executing**
2. So, in a closure, a function is defined along with the environment.
3. In Python, this is done by nesting a function inside the encapsulating function and then returning the underlying function.
4. **Closures can avoid the use of global values and provides some form of data hiding. It can also provide an object oriented solution to the problem**.
5. This happens because we can can’t directly call the nested function.
6. When there are few methods (one method in most cases) to be implemented in a class, closures can provide an alternate and more elegant solutions. But when the number of attributes and methods get larger, better implement a class.

def make\_multiplier\_of(n):

def multiplier(x):

return x \* n

return multiplier # return a function without providing any parenthesis.

# 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)))

1. Decorators:

* **a decorator is a function that takes another function and extends the behavior of the latter function without explicitly modifying it**

1. import time
2. def timing\_function(some\_function):
3. """
4. Outputs the time a function takes
5. to execute.
6. """
7. def wrapper():
8. t1 = time.time()
9. some\_function()
10. t2 = time.time()
11. return "Time it took to run the function: " + str((t2 - t1)) + "\n"
12. return wrapper
13. @timing\_function
14. def my\_function():
15. num\_list = []
16. for num in (range(0, 10000)):
17. num\_list.append(num)
18. print("\nSum of all the numbers: " + str((sum(num\_list))))
19. print(my\_function())

* somefunction = someotherfunction(somefunction):

can be written as @someotherfunction above the somefunction.

**13. Generators:**

a) Eager vs Lazy:

* Eager: When some computation is happening and it returns the entire result to us only after completion.
* This is not always desirable, for example, if some computation requires a lot of time, then we have to wait for that time to finish.
* There are scenarios when we don’t want need all results at the same time, and we might need them one by one.
* Generators use ‘yield’ keyword to return the result back to the caller.

2. Control transfer:

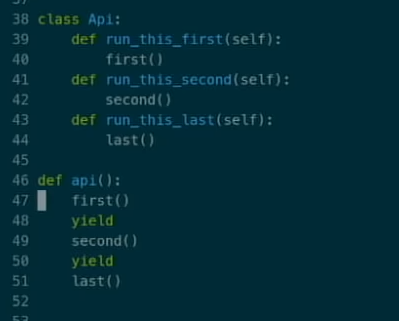
- With Generators the control of the flow is being transferred back to the caller when the yield keyword returns.

- So this idea of co-routines is also a major feature of generators.

- For example, there is an API with 3 sets of methods, and API creator wants us to call those methods in some specific order and mentions that clearly in the documentation.

- Its only documented and it not stopping us from calling the methods in any different order that can break our code.

- In this case, we use generators such that the methods are always called in desired order:



* Generators are also used in Context managers to enable sequencing of entry and exit method.

**14. Context Manager:**

* In certain scenarios, we need to do some setup/teardown or initial/final actions.
* ‘with’ keyword to open a file is an example of context manager.
* To implement a context manager we need to implement the ‘\_\_enter\_\_’ and ‘\_\_exit\_\_’ dunder methods.

15. Exception Handling:

- There can be cases when there is an error in our program the stops the execution.

- Sometimes we want to accept these errors and don’t stop the execution. This is called handling the exception.

- In python, we use **‘try except’** block for handling exception.

try :

#some code that may raise exception

except typeofexception1:

#some code to handle the exception raised

finally:

#block of code that’s always executed. Used to do

some clean up actions.

- Finally if mentioned, gets always executed.

- A try block can have multiple except: blocks to handle different types of exception.

- using only ‘except:’ will handle all the exceptions:

- writing except typeofexception: will handle any specific exception.

- **We can also have ‘else’ with the try catch. It gets executed when there is no exception. Syntax:**

try:

except:

else:

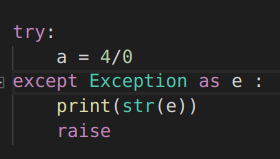
**Raise keyword**

* Raise keyword has two main uses:

a) To raise your own errors:



b) To raise (or re-raise) the current exception in the exception handler so that it can be reraised in the call stack.



**Python errors**:

* Name error: When we are using some thing that does not exist. (‘Name <> is not defined)
* Type error : Giving a wrong argument to a function or we are trying to do something that is not supported.
* for example: len(5) will give- TypeError: object of type 'int' has no len()
* Syntax error :
* Value error : The value that we wanted is not available.
* keyerror: fetching value for dictionary using a key that does not exist.
* Attribute Error: ‘string’.contains(’t’)
* AttributeError: 'str' object has no attribute 'contains'

Typeerror vs ValueError:

* int(‘a’) : is expecting a string. Its getting the right type, but it can’t convert the value ‘a’ to an int. So its an value error.
* a = {‘name’: ‘Santosh’}
* int(a) : will give type error. Because int is expecting a string not a dict.
* Int(‘a’) : will give value error. Because the type is correct, that is string, it int can’t convert the value ‘a’ into an ‘int’.

1. Difference Between Python 2 and Python 3:

a) Divison operator: returns integer in python 2 and float in python 3.

- use \\ in python3 to get a integer out of division.

1. Print function.
2. Range and xrange() funciton.
3. Error Handling: python 3 need to use ‘as’ keyword’ when expecting an error in try:except block.
4. Python 3: all the classes implicity inherits from the object class. In python2, we have old class style classes that donot inhert from object class and we can use new class style to make them inhert from object base class explicitly.
5. String interpolation (python3.6)- We can use following format starting python 3.6

name = ‘santosh’

f’my name is {name}’

where we can add python expressions inside the curly braces.

1. Type annotations:

* We can provide the type hints before variables or return type, to help the user code to run into any run time error due to types.
* Helps to develop better and faster
* Helps the users of your code.
* Easy documentation without using any doc string.

Reference:

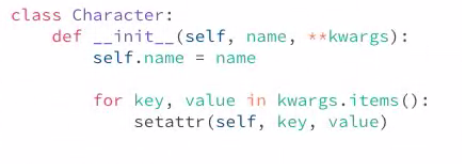
https://medium.com/@ageitgey/learn-how-to-use-static-type-checking-in-python-3-6-in-10-minutes-12c86d72677b

1. Monkey patching:

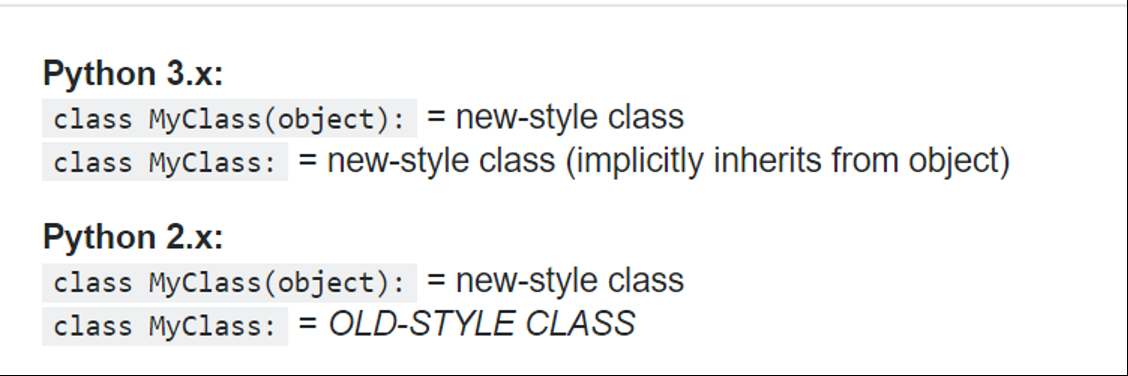
* A MonkeyPatch is a piece of Python code which extends or modifies other code at runtime

1. setAttr method:

-this method is used to dynamically create attribute for a method based on the kwargs:



1. Old class Vs New Class



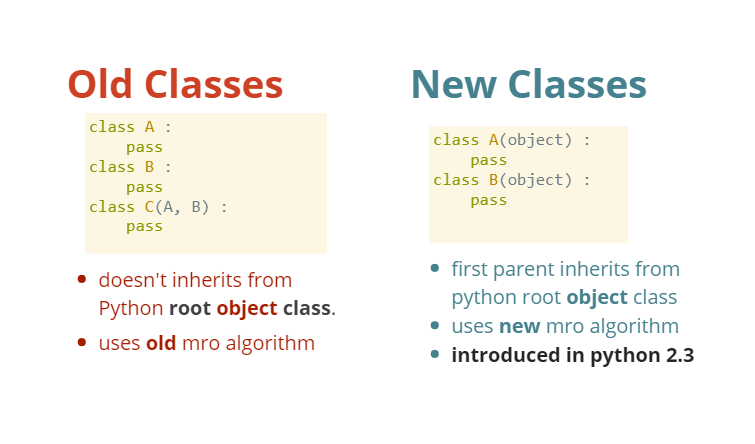
In New Style classes:

1. Super method in added.
2. MRO is changed.
3. Class can’t be raised unless derived from Exception class.
4. Descriptors are added ???
5. Slots are added ???

MRO : Method Resolution Order:

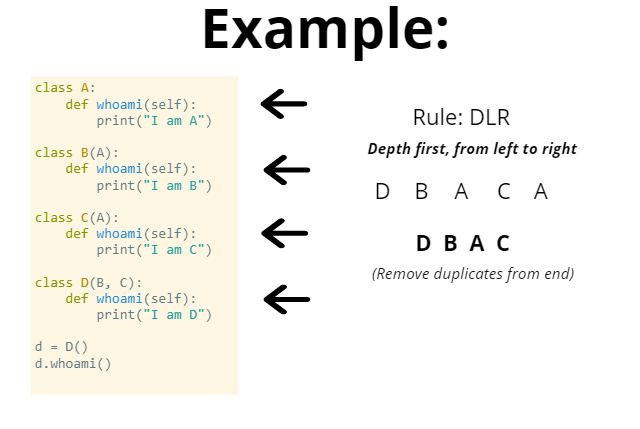
* defines the **class search path** used by Python **to search for the right method/variable to use** in classes having multi-inheritance.

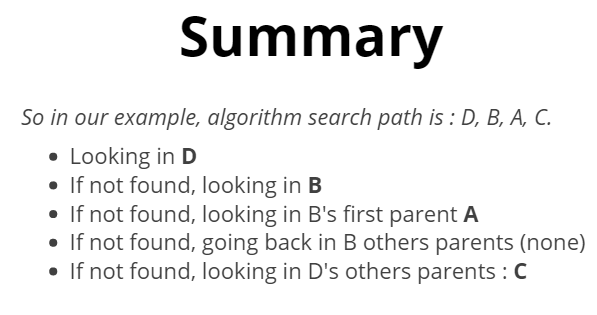
Python old Classes VS New classes



* In python 3, the classes inherits from Object class implicitly.
* Python 2:

Depth First From Left to Right.

- 

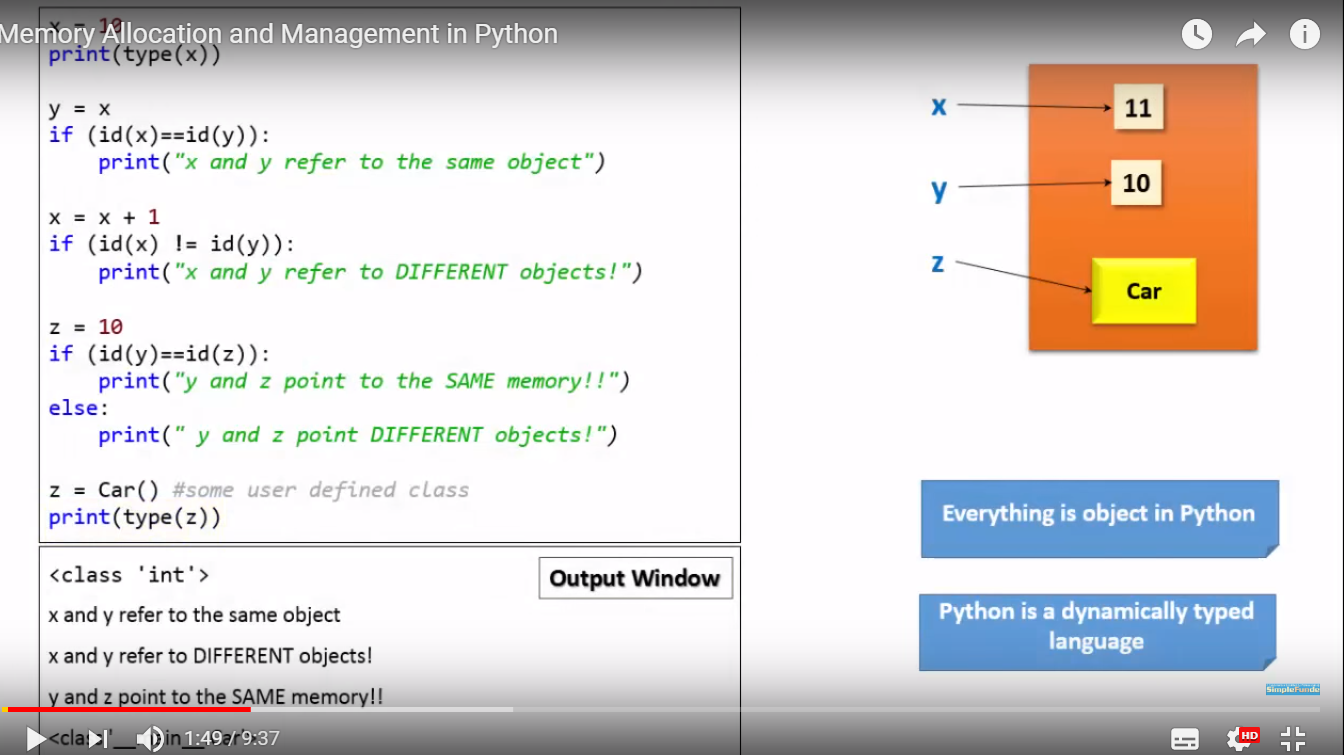


**Python 2 MRO :**

class A1():  
#      def who\_am\_i(self):  
#          print("I am a A1")  
    pass  
  
class A2():  
     def who\_am\_i(self):  
         print("I am a A2")  
  
class A3():  
     def who\_am\_i(self):  
         print("I am a A3")  
  
class B(A1, A2):  
#     def who\_am\_i(self):  
#         print("I am a B")  
    pass  
  
class C(A3):  
    def who\_am\_i(self):  
        print("I am a C")  
  
class D(B,C):  
#     def who\_am\_i(self):  
#         print("I am a D")  
    pass  
  
d1 = D()  
d1.who\_am\_i()

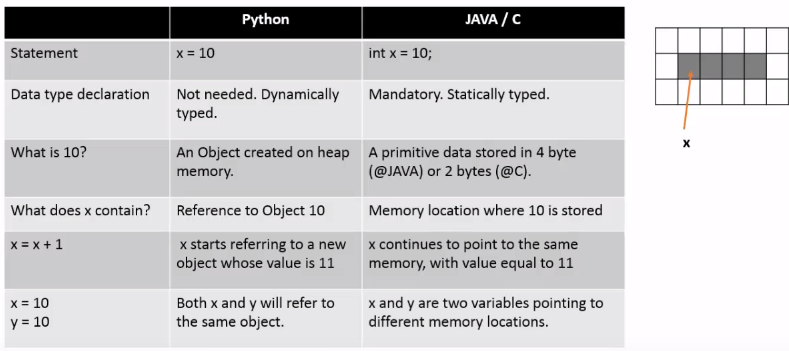
The search path when invoking d1.who\_am\_i() is : D, B, A1, A2, C, A3

1. **Python Memory Management:**

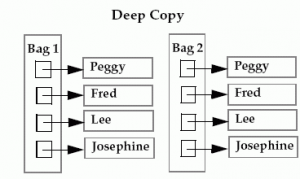
****

* Methods and variables are created in Stack memory.
* **Objects and instance variables are created in Heap memory.**
* A new stack frame is created on invocation of a function/method.
* These stack frames are destroyed as soon as the function returns.
* Garbage collector is the mechanism to clean up the dead objects.
* An object is dead when there are no references pointing to it.
* **Python keeps the reference count for each object, when that reference count is 0, the object is deleted.**

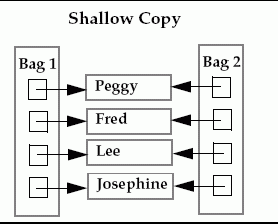
**Differnece between Python and Java memory allocation:**

****

1. **Deep Copy vs Shallow Copy:**

[](http://cdncontribute.geeksforgeeks.org/wp-content/uploads/Deep.png)

* In case of deep copy, a copy of object is copied in other object. It means that **any changes** made to a copy of object **do not reflect** in the original object.
* In python, this is implemented using “**deepcopy()**” function.

[](http://cdncontribute.geeksforgeeks.org/wp-content/uploads/Shallow.png)

* In case of shallow copy, a reference of object is copied in other object. It means that **any changes**made to a copy of object **do reflect** in the original object.
* In python, this is implemented using “**copy()**” function.

Example:

Import copy

a = [1, 2, [3, 4]]

Shallow = copy.copy(a)

Deep = copy.deepcopy(a)

a[2][0] = 4

a[0] = 0

a 🡪 [1, 2, [4, 4]]

b 🡪 [1, 2, [4, 4]]

c 🡪 [1, 2, [4, 4]]

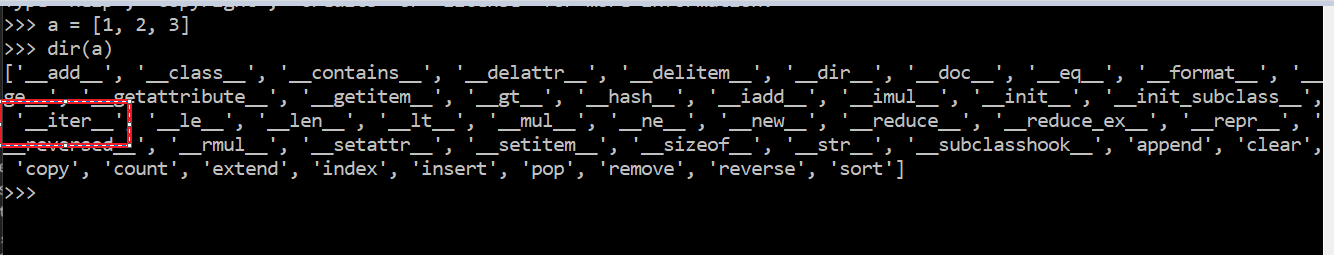
The difference between shallow and deep copying is only relevant for compound objects (objects that contain other objects, like lists or class instances):

* A *shallow copy* constructs a new compound object and then (to the extent possible) inserts *references* into it to the objects found in the original.
* A *deep copy* constructs a new compound object and then, recursively, inserts *copies* into it of the objects found in the original.
* **20: Iterators vs Iterables**.

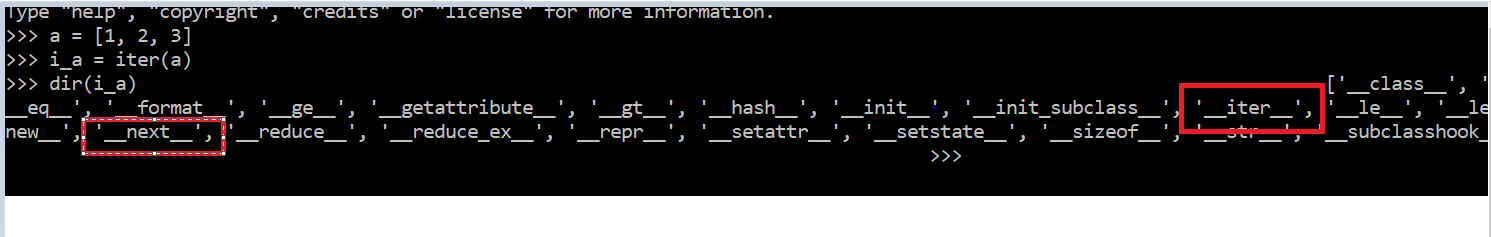
a) Iterable:

* something that can be looped over. For example a list.
* - If something is iterable (can be looped over), then it needs to have a special method called \_\_iter\_\_()
* Nums = [1, 2, 3]
* For num in Nums:
* Print(num)
* - Here nums is a list and is iterable

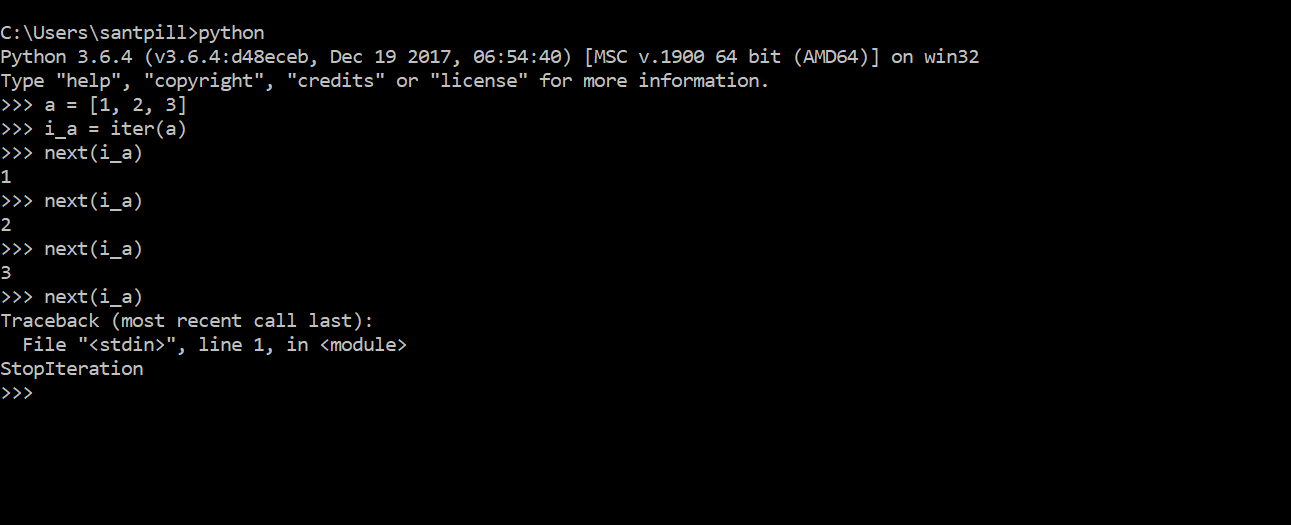
- dir(Nums): will show us all the methods in the list object. And \_\_iter\_\_ will be one of them.

* 
* Note: next() method calls the \_\_next\_\_ method and iter() method calls the \_\_iter\_\_ method in the background.
* - Running a \_\_iter\_\_ method on a list returns an iterator.
* - Running \_\_iter\_\_ on an iterator returns the same object.
* - Running \_\_next\_\_ method on an iterator returns the next value.
* - Running \_\_next\_\_ on a list returns an error (as list is not an iterator and does not have next() method)

b) Iterator:

* - **An object with a state so that it remembers it value.**
* - Iterators get the next value using the \_\_next\_\_ method.
* - a = [1, 2, 3]
* iter(a) : will return an iterator.
* next(a) : will return an error because list ‘a’ is not an iterator.
* - passing an object into a Next method will call the \_\_next\_\_ method for that object class. Since lists donot have the dunder next method, it fails.
* 

- Iterator remembers the previous value and next value can be obtained using the next function. When there is no next value, it throws a stop iteration exception.

* 

- Iterators can only go forward. There is no going backward, resetting some value etc

21**: Python Multithreading and its limitations:**

1. Limitations:
   * Standard implementation of python written in C is known as CPython.
   * This cPython interpreter uses GIL (Global Interpreter Lock)
   * GIL is a the mechanism used by computer-language interpreters to synchronize the execution of threads so that only one native thread can be executed at one time.
   * So an interpreter that uses GIL allows only one thread to execute at a time, even if we run a multi core processor.
2. What is GIL?
   * It’s **a mutex or a lock** that allows only one thread to hold the control of a python interpreter.
   * This means that only one thread can be in a state of execution at any given time, even with a multithreaded architecture with more than one CPU core.
3. Why was GIL used in Python?
   * Python uses reference count mechanism for memory management.
   * This reference count variable needed protection from race conditions, where two threads increase or decrease its value simultaneously. So if this happens, it can **cause leaked memory that is never released or even worse, incorrectly release the memory while reference to that object still exists**. This can cause crashes or other weird bugs in python program.
   * One solution for this could have been to add locks for each data structure that are shared across the threads so that they are not modified inconsistently.
   * But adding locks to each object or multiple objects will result in multiple locks that can :
   1. Cause deadlock.
   2. Decreased performance due to repeated acquisition and release of locks.
   * **GIL is a single lock on the interpreter itself.**
   * **This means that for executing any python bytecode, this interpreter locks needs to be acquired**.
   * This prevents deadlock, does not impose a performance overhead.
   * But this also makes any CPU bound python program single threaded.
4. Impact on multi-thread Programs:
   * There are two type of programs:

1. CPU bound:

* + These push the limit of CPU.
  + For example, mathematical computation like matrix multiplication, searching, image processing etc.

2.I/O bound:

- Programs that spend time waiting on input/ output which can come from a user, file, database, network, etc.

- **GIL does not have any major impact on I/O bound multithreaded programs as lock is shared between the threads as they are waiting for IO.**

- **GIL will make a multiply threaded CPU bound task into single thread and will also increase the execution time (compared to single threaded one) due to lock acquire and release overhead**.

1. Why GIL hasn’t been removed?
   * Several attempts has been made in the past but that causes many backward compatibility issues.
   * It also breaks the existing C extension that heavily rely on solutions that GIL gives.
   * Removing GIL in python3 would have made in slower than Python2 when compared to single threaded programs.
2. How Python3 has improved on GIL:
   * **Python had a mechanism that forced threads to release GIL after a fixed interval of continuous use and if no body else acquired the GIL, the same thread will continue to use it.**
   * Due to this mechanism, most of the time CPU bound threads would acquire the lock(GIL) before other threads could acquire it.
   * As a result, I/O bound threads would starve by not giving them a chance to acquire the GIL from CPU based threads.
   * This problem was fixed in Python 3.2 in 2009 by Antoine Pitrou who added a mechanism of looking at the number of GIL acquisition requests by other threads that got dropped and not allowing the current thread to reacquire GIL before other threads got a chance to run.
3. How to deal with GIL?

1. **Use Multi processing**:

* + Popular way is the use mult-processing the uses multiple processes instead of multiple-threads.
  + **Each python process gets its own python interpreter and memory space so GIL won’t be a problem.**
  + Multiprocessing module in python can used to achieve this.

2.Use other interpreters:

- GIL is only used in Cpython. We can use Jython, pypy or IronPython if our programs with its libraries are supported by these interpreters.

**Some basic concepts:**

**What is Python**:

* Interpreted:
* Object Oriented (but not completely)
* High level
* Dynamic Typing

**Why Python**:

* Rapid Application Development
* Simple
* Easy to learn
* Readable
* Reduces cost of program maintenance Vs Java:
* slower to execute compared to java why?
* Takes less time to develop
* Programs are 3-5 times shorter than equivalent java programs

**Python shell**:

* use help() to get details about inbuilt things in python.
* Python scripts are written in text files with extension .py

BOOLEAN

* number 0 in any form where being empty string or object are false.
* so anything that is empty or 0 is false
* everything else is true.
* bool(None) is also False
* We don’t have === in python like we have in javascript

STRING

* sequence of characters.
* contains a-z, 0-9, @#$%$ etc.
* it should defined in double or single quotes.
* use escape sequence backslash ‘\’ to ignore something after it.
* We can use backslash to print the string in the new line in the code (not in the output though).

text = “some text \

in next line”

* Built in methods for string:
* len(string)
* string.lower()
* string.upper()
* str(something) : converts something in string form
* string.replace(replace, with , count)
* string[1:6] - to get the substring from a string.
* string[0basedStartRange:1basedEndRange:Steps] - steps refer to the skip in the getting the substring.
* print (“SantoshPILLAI”[0:8:2])
* output: Snoh
* string[StartIndex:] - Starts printing after start index till the end as no end index is provided.
* string[-1:] - minus indexing starts from the end.
* string[::] - prints the string as it is.
* string[::-1] prints the reverse of the string.
* String formating:
* There are 2 ways:
* print ("My name is " + text3) print ("My name is %s and previously it was %s" %(text3, text2))

here %s is like a place holder that will be replaced by values in %() in the same order.

* print(“My name is {} and previously it was {}”.format(text3, text2))
* Concatenation:
* We can add 2 strings using a + operator.
* If any of the operand is not a string then we have to convert it into a string using ‘str’ function else we get a type error.
* In python a string is any bit of characters between quotes
* Quotes:
  + “test” is valid string
  + ‘test’ is also a valid string.
  + We can use both single and double strings to represent a variable.
  + “He’s right” is valid string.
  + ‘He’s right’ will give error. So we can use the escape character to let python ignore a quote.
* Strings can also be used in triple quotes.
* “”” this is string”””
* ‘’’ this is string ‘’’
* The str() function to covert anything to a string.
* We can do multiply operation on strings for some formatting.
* “—“ \* 20
  + we can add dynamic values to string using format function
* print ( "this is {} {}" .format(4, 10))
  + Strings have to start and stop with the same quote symbol, either single (') or double (") or triple (''' or """) quotes.
* You also don't want to use that same quote symbol inside the string unless you escape with with backslash (\).
* Strings can be combined with the plus sign (+) which is known as concatenation.
* Strings can also be combined using string formatting which uses two braces ({}) as placeholders and the .format() method. You provide a bit of data to .format() for every placeholder in your string. For example:“My {} is {}".format("name", "Kenneth")
* This string has two placeholders so we have to give two new things to .format().
* You can also multiply strings by an integer. This will result in the string's content being repeated as many times as the value of the integer. "hi" \* 3 would create "hihihi".
* And, finally, you can create a string from another value by using the str() function.
* We can not something from a string directly as strings are immutable. So convert string to a list, delete the item and then convert back to a string
* IMPORTANT:
  + Strings are immutable.
  + s = ‘Santosh’ s[0] = ‘b’ will give error.
* Reason for string immutability:
  + One is performance: knowing that a string is immutable makes it easy to lay it out at construction time — fixed and unchanging storage requirements.
  + String can be used in Dict as keys.
  + Makes working with strings threadsafe.

Why is python slower than Java?

What is the init.py file?

* + Init.py file in a directory tells the python interpreter to use this directory as a python package directory.
  + This helps us in importing or using modules across different python directories.
  + For example:
* package**/**
* \_\_init\_\_.py
* file.py
* file2.py
* file3.py
* subpackage**/**
* \_\_init\_\_.py
* submodule1.py
* submodule2.py

because of using init.py files, file.py can use submodule1.py, present in a different packages,

What goes inside \_\_init\_\_.py

\_\_init\_\_.py can be an empty file but it is often used to perform setup needed for the package(import things, load things into path, etc).

- Advanced Data type:

Python Collections:

————

**List:** similar to arrays

* **data type to store more than one value in one variable**
* **Element are stored in a order called index of the elements**
* elements are added inside square brackets
* car[“maruti” , “Honda”, “BMW”]
* changing values in a list
* car[0] = “Tata”

List built- in methods:

* len(cars) : gets the length of the list
* cars.append(“benz”) : add item at the end.
* cars.insert(index, “suzuki”): add item at a certain position. We need to know the index in advance.
* cars.index(“Honda”) : get index position of an item.
* cars.pop() : removes the last element of the list and gives it back to us. It either takes no argument (then remove the last element) or we can provide a specific index to it.
* del cars[index] - removes the item based on the index
* cars.remove(“Tata”) : removes the given item from the list.
* cars[0basedStartIndex:1basedendIndex] : to slice the list.
* Slicing can be used with both list and strings
* cars.sort() : sorts the list in some specific order based on the type of data in the list. All types should be same for using this method.

Extending our list:

* We can combine out list just like combining two Strings
* our\_list = [1, 2, 3, 4,]
* **our\_list = our\_list + [5, 6, 7]**
  + **[1, 2, 3, 4, 5, 6, 7]**
* our\_list = our\_list.append([8, 9, 10])
  + [1, 2, 3, 4, 5, 6, 7, [8, 9, 10]]

extends method:

* using + sign to add lists is not that clean. So ‘extend’ method is a better approach. Its faster for bigger lists. Extend method extends the list by appending elements from the iterable.
* our\_list = [1, 2, 3]
* our\_list = our\_list.extend(range (4, 10000))
* Inserting items:
* We can use insert method to insert an item into a list at a specific location.
* our\_list = [1, 2, 3]
* our\_list.insert(0, 0) # insert(index, value)
* [0, 1, 2, 3]

SLICING:

* [start:stop:step]
* Copy list into another:
* myList2 = [5, 4, 3, 2, 1]

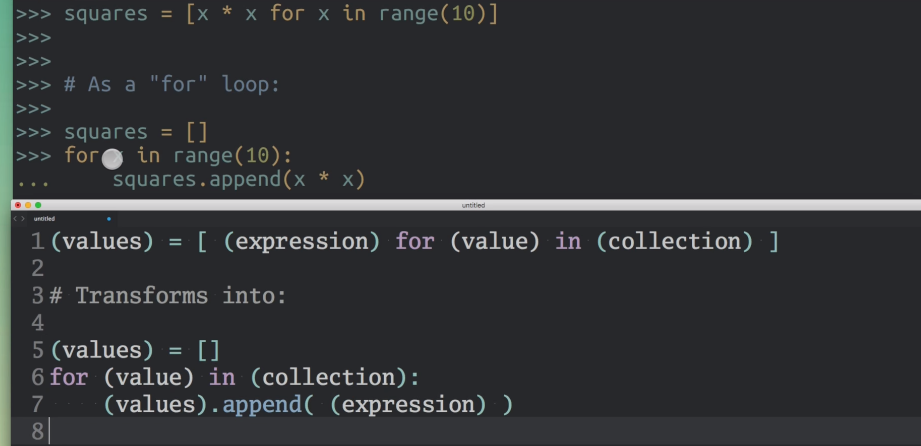
myList3 = myList[:] #creates a copy of the myList so changes does not reflect on other list.

myList3.sort()

print (myList2)

print (myList3)

* Slicing limited part of the data:
* Deleting or Replacing slices:
* myList = [1, 2, 3, ‘a’, ‘b’, ‘c’, 7]
* myList[3:6] = [4, 5, 6]
* When replacing a slice with another iterable, they need NOT be the same size.
* >>>myList = [1, 2, 3, "a", "b" , "c", 7]
* >>>myList[3:6] = [4, 5, 6]
* >>>myList
* [1, 2, 3, 4, 5, 6, 7]
* >>>myList[3:6] = [9, 10]
* >>>myList
* [1, 2, 3, 9, 10, 7]
* List comprehension:
  + A shorthand for creating a list.



* + We can also add conditional statements inside a list comprehension.

squares = [x \* x for x in range(10) if x % 2 == 0]

**Dictionary:**

In python dictionaries are hashtable implemetation.

* data type to store more than one value in a variable.
* elements are added inside curly braces.
* elements are added in key value pairs {key:value}
* there is no indexed. elements are not ordered and can’t be accessed via index (like in list).
* Elements can be accessed based on mapping using the keys.

Example.

myDict1={"Name":"Santosh" , "Age" : "26"}

* Nesting dictionary:

employee= {‘emp1’:{‘name’: ‘Santosh’, ‘age’:26} ,‘emp2’ :{‘name’ : ‘Test’ , ‘age’ : 80}}

print (employee)

print (employee['emp1']) print (employee['emp1']['name'])

* Dictionary builtin Methods:

1. **myDict1.keys()** - gets all the keys inside the the dictionary as a list
2. del myDict[“key\_Name”]- deletes a key value pair from the dictionary
3. myDict[“key”] = “value” - adding/updating a single key in dictionary.
4. myDict.update( {key1: value1 , key2: value 2} ) - adds/updating multiple key value pairs
5. myDict1.values() - gets all the values inside the dictionary as a list
6. myDict1.items() - gets all the items inside the dictionary as tuple.
7. myDict1.clear() - clears the dictionary for all its elements.
8. myDict1.pop(“Age”) - pops up the key value pair from the dictionary based on the key provided as argument.

* Unpacking a dictionary:
  + we can unpack a dictionary that is get data out of using \*\* symbol
  + for example

myDict2 = { 'name' : 'Santosh' , 'age' : 26 , 'profession' : 'Engineer'}  print (" My name is {name}. I am {age} years old and I am an {profession} by profession ".format(\*\*myDict2))

* Looping through a dictionary

Unlike strings and lists, dictionary does not have a set order

We can loop over both values and keys present inside a dictionary

my\_dict = { ‘key1’: ‘value1, …..}

for keys in my\_dic.keys():

print(keys) —prints the keys

print(my\_dict[keys]) — prints the values based on the keys

for value in my\_dict.values():

print value — prints the values.

for key, value in my\_dict.items():

* We can’t slice a dictionary as values are not ordered.
* Dict comprehension:
  + Shorthand for creating a dict
  + b = { x : x\*x for x in range(10)}
  + We use {}, and expression should return a key-value thing.

—————————

**Tuple: similar to List but they can’t be changed.**

* similar to lists but are immutable
* values can’t be changed.
* values are entered in round brackets.
* Memory efficient.
* my\_tuple = (1, 2, 3)
* its not the parenthesis that make a tuple, a tuple. its the commas
* my\_tuple = 1, 2, 3 is also a tuple.
* parenthesis make it easier to recognise tuples in the code.

**Tuple builtin methods:**

my\_tuple.index(1) : to get the index.

my\_tuple[1:

Converting list to tuple using tuple method:

my\_tuple = tuple([1,2, 3])

**Uses of tuplet**

* **tuples have fixed size and memory so it reduces the memory usage.**
* **We can do simultaneous assignment using tuples a, b = 1, 2**

This is also related to how packing and unpacking works.

Common use case is swapping variables:

a, b = b , a

* we can use tuple as return values of function to return multiple values at the same time.
* We can unpack a tuple using \* symbol
* **Tuples are immutable, that is we can not change the value inside it. But we can still add a mutable object inside a tuple and then change that.**

mytup = (1, ‘string’ , [1, 2, 3])

mytup[2][1]= 4

mytup =

(1, ’string’ [1, 4, 3])

**We can update a mutable value inside a string but not delete it completely.**

**Enumerate:**

* This methods takes an iterable and gives us each value in it and its index position.

 for index, value in enumerate(range(1, 100)):

print("{} {}".format(index+1, value))

**Sets:Collection of unique elements.**

* Created using curly braces.- {1, 2, 3} is a Set.
* A blank set is treated as a dictionary. So in that case we need to use a set method.
* Adding to Set. (add)
* Sets are mutable like list so we can add to it using:

a= {1, 2, 3}

a.add(4)

duplicate elements are not added while adding.

* Updating set: (update)

- We can update the Set with more values using update method:

a.update({5, 6} , {7, 8})

{1, 2, 3, 4, 5, 6, 7, 8 }

* Deleting from Set: (remove , discard, pop)

- We can delete using remove() method. It takes the item that we want to remove.

a.remove(8)

- if the item we want to remove does not exist, then we will get a key error.

- discard method also removes the value, but it does not give any key error if the value is not present.

pop method removes the last element and returns it.

* **We can also perform following math operations on Sets**

**Union: Returns all the elements in both the sets (discarding the duplication).**

set1 = {1, 2, 3}

set2 = {3, 4, 5}

set1.union(set2)

0r

set1 | set2

{1, 2, 3, 4, 5}

**- Union is similar to update but it does not change the original set.**

**- Difference: to find elements in one set that is not present in the other set.**

set1.difference(set2) —> {1, 2, 3}

set2.difference(set1) —> {3, 4, 5}

The difference operation is - symbol

set1 - set2 —> {1, 2}

set2- set1 —> {4, 5}

**Symmetric difference : to get combination of elements that are present in both the sets and discarding the common elements.**

set1.symmetric\_difference(set2) —> {1, 2, 4, 5}

set2 ^ set1 —> {1, 2, 4, 5}

Intersection: Get the element present in both the sets.

Set1.intersection(set2) —> {3}

set2 & set1 —> {3}

Set Comprehension:

* + Shorthand for creating a set quickly.
  + a = { x for x in range(10) }
  + using {} for creating a set.
  + The expression should not return a key-value type. It should return a single value.

——————

**Comparators:**

* == : refers to equality and is know as equality operator.
* != : not equal to
* < : less than
  + : greater than
* <= : less than equal to
* >= : greater than equal to.

**Boolean comparators**

* and : returns true if both are true. If the first is false, then it returns false without executing the second operator.
* or : return true of any of the expression is true. if first is true then it does not evaluate the second expression.
* Not: Returns the opposite the result returned by the expression.
* The order of precedence is Not or and.

**Conditional Statement:**

* if ,elif and else
* in Keyword

Checks for containment. That is whether a certain value is another value.

2 in test[1, 2, 3]

It returns a boolean.

This is mostly used with if statement and in for loop

It can also be used as :

if something not in something else

if not something in something else

**Taking user input:**

We can take inputs from the user using the input method.

age = int(input(“whats your age? “)

**Looping statements:**

while

for

- for something in SomethingElse

here something is variable created on the fly which takes the values inside ‘SomethingElse

break: to break out of the loop

continue: to break of the current iteration.

While- else

**Iterating over multiple list:**

This can be done using built in method ‘zip’ . It works in pair. so it will stop after completing the shorter list.

l1=[1, 2, 3]

l2 = [4, 5, 6, 7, 8, 9, 10]

for a , b in zip(l1, l2):

print (a)

print (b)

out: 1 2 3 4 5 6

It loops three time only as that is the limit of the first loop.

**Rang function**:

Built in function

creates a sequence of number.

this is not saved in memory so there is not load on our computer.So this can be used to create a list of millions of number without affecting the computer memory

———————————————————

Methods:

A function used inside a class is a method.

building blocks in any programming language

a reusable piece of code that does some task.

syntax : def methodname(arguments) :

def stands for define.

syntax should end with a colon.

Naming convention: use all small case separated by underscore or use camel casing

Always write the doc code about what the method does.

doc code is written as the first thing inside the method. just type “ “ “ and then hit enter and it will be done automatically in pycharm .

Positional parameters:

These are optional parameters and are used to assign default values when no values are passed in the method call.

def sum (n1=2, n2=3)

we can simple call the method without any arguments as we are using optional parameters in the method call.

sum()

We can also change the positions using positional parameters using positional parameters.

sum (n2=3 , n1=2)

We can also change the number of arguments passed like.

sum (n2=3)

Some built in functions:

max() - reruns the maximum from the given argument

min() - returns the minimum form the given arguments

abs() - returns the absolute value of the given value

type() - returns the type of the given value, whether its a string or list or something else.

We can pass multiple arguments to a method using following syntax:

def someMethod(\*args):

someMethod(1, 2, 3, 4, 5)

Variable Scope:

A variable declared inside a method can’t be used outside the method.

If two variables have same name and one has global and other has local scope, then changing the locally scoped variable won’t change the global one.

use keyword ‘global’ in side the method, to ensure that you are using the global variable and in this cases, the changes made the variable in sided the method will affect the global variable as well.

Range Function:

When:

Used when want to perform an action X number of times, where you may or may not care about the index.

​

ADVANCED CONCEPTS:

* + \_\_name\_\_ is a special variable that refers to the current namespace.
  + To prevent code from being executed when you import a module, put it into a function or class, or inside following if condition:

if \_\_\_name\_\_== “\_\_main\_\_”

Object Oriented Programming in Python:

* + Classes and Objects

Class are are like a block of statements that hold other things like properties and methods.

class Animal:

name = “Dog”

color = “Black

* + We can create an object or any instance of the class simply by

my\_obj = Animal()

* + To make the classes useful we write functions inside the classes, and the functions inside a class are called as methods.

class Animal :

name = “Dog”

sound = “bark”

def sound (self){

return self.sound.uppercase()

animal = Animal()

animal.sound()

>>> BARK

Every method in class, at very least, takes one argument called as ‘Self’, this is mendatory

‘Self’ represents the instance on which we are calling this method on. We don’t have to pass it when calling the function, but it should be mentioned in the method definition.

Variables defined inside a class but outside any method are class variables. All the objects have access to same copy of this method.

Variables defined inside a class method are called as instance methods.

Dunderinit( \_\_init\_\_ )

Lots of times we want our object to have certain values for their attributes when we start. This is done by \_\_init\_\_

\_\_init\_\_ is called when we create an object of our class.

This is special python methods.

this is similar to constructors in Java.

it is used the do some initialisation when class object is created ( object name = ClassName():)

\*\*kwargs: helps us to unpack the parameters send to us as a dictionary and then get the value using the key name

to get value: kwargs[key\_name]

Dunderstr( \_\_str\_\_):

This is a method that gets called when over something is converted in to a string.

Inheritance:

Means to have a parent class with subclasses, where subclass can use attributes (properties and methods of the parent class)

Syntax:

class Child (Parent):

Overloading and Overriding in python:

* + Overloading: not supported in the python.

But we can perform overloading using variable arguments as the method parameter.

Overriding:supported in python.

overridden method

Built in modules

These are just the external python files that we can include in our project in order to use their methods.

syntax

import math - this imports the complete math module into our program.

from math import sort - This imports the square root method from the math module.

Writing to a file

To read/write data to a file we need to open the file first:

my\_file = open("text1.txt" , “mode”)

We can have following ‘Modes’ to manipulate a file:

w - Write only mode

r - read only mode

r+ - read and write mode

a - append mode,

To write to a file:

my\_file.write(“Some data”)

once we are done with the file, its important to close

my\_file.close()

This close() method invokes the file objects exit() method to close it.

Reading a file:

myfile = open("text1.txt" , 'r') print(str(myfile.read())) my\_file.close()

Use readline() method to read the file line by line.

It reads only one line at a time.

myfile = open("text1.txt" , 'r') print(str(myfile.readline())) my\_file.close()

Important :Its really important that we close the file. For example, if file is not closed after last manipulation, and then we try to open it again to read, then it won’t read.

With/As keywords

using with-as keyword we can read and write into a file without having to close it manually.

Example

data = "some raw data to be displayed"  with open("text1.txt" , 'w') as with\_as\_write: with\_as\_write.write(data)  with open("text1.txt" , 'r') as with\_as\_read:  print (str(with\_as\_read.read()))

Regular Expressions:

Used to match patters with text

‘import re ‘ library for doing regex in python

re.match(r’\w’, data) - matches with the first word of the content.

re.search(r’\w’ , data) - searches in the first line.

.findall(pattern, text, flags) - Finds all non-overlapping occurrences of the pattern in the text.

• re.IGNORECASE or re.I - flag to make a search case-insensitive. re.match('A', 'apple', re.I) would find the 'a' in 'apple'.

• re.VERBOSE or re.X - flag that allows regular expressions to span multiple lines and contain (ignored) whitespace and comments. We can still use a explicit space in the set

escape characters:

• \w - matches an Unicode word character. That's any letter, uppercase or lowercase, numbers, and the underscore character. In "new-releases-204", \w would match each of the letters in "new" and "releases" and the numbers 2, 0, and 4. It wouldn't match the hyphens.

• \W - is the opposite to \w and matches anything that isn't an Unicode word character. In "new-releases-204", \W would only match the hyphens.

• \s - matches whitespace, so spaces, tabs, newlines, etc.

• \S - matches everything that isn't whitespace.

• \d - is how we match any number from 0 to 9

• \D - matches anything that isn't a number.

• \b - matches word boundaries. What's a word boundary? It's the edges of word, defined by white space or the edges of the string.

• \B - matches anything that isn't the edges of a word.

Two other escape characters that we didn't cover in the video are \A and \Z. These match the beginning and the end of the string, respectively. As we'll learn later, though, ^ and $ are more commonly used and usually what you actually want.

Counts: To make more effective and readable patterns

\w{3} - matches any three word characters in a row.

\w{,3} - matches 0, 1, 2, or 3 word characters in a row.

\w{3,} - matches 3 or more word characters in a row. There's no upper limit.

\w{3, 5} - matches 3, 4, or 5 word characters in a row.

\w? - matches 0 or 1 word characters.

\w\* - matches 0 or more word characters. Since there is no upper limit, this is, effectively, infinite word characters.

\w+ - matches 1 or more word characters. Like \*, it has no upper limit, but it has to occur at least once.

Sets: Matching exact characters and etc:

• [abc] - this is a set of the characters 'a', 'b', and 'c'. It'll match any of those characters, in any order, but only once each.

• [a-z], [A-Z], or [a-zA-Z] - ranges that'll match any/all letters in the English alphabet in lowercase, uppercase, or both upper and lowercases.

• [0-9] - range that'll match any number from 0 to 9. You can change the ends to restrict the set.

re.INGNORECASE or re.I tag is added as the third argument in the method to ignore the cases.

Negation: leaving a particular pattern that we don’t want.

[^abc] - a set that will not match, and, in fact, exclude, the letters 'a', 'b', and 'c'.

Groups: are defined using parenthesis ()

• ([abc]) - creates a group that contains a set for the letters 'a', 'b', and 'c'. This could be later accessed from the Match object as .group(1)

• (?P<name>[abc]) - creates a named group that contains a set for the letters 'a', 'b', and 'c'. This could later be accessed from the Match object as .group('name').

• .groups() - method to show all of the groups on a Match object.

• re.MULTILINE or re.M - flag to make a pattern regard lines in your text as the beginning or end of a string.

• ^ - specifies, in a pattern, the beginning of the string.

• $ - specifies, in a pattern, the end of the string.

Advanced Regex:

• re.compile(pattern, flags) - method to pre-compile and save a regular expression pattern, and any associated flags, for later use.

• .groupdict() - method to generate a dictionary from a Match object's groups. The keys will be the group names. The values will be the results of the patterns in the group.

• re.finditer() - method to generate an iterable from the non-overlapping matches of a regular expression. Very handy for for loops.

• .group() - method to access the content of a group. 0 or none is the entire match. 1 through how ever many groups you have will get that group. Or use a group's name to get it if you're using named groups.

​

Python Testing

In development we use an approach called TDD, Test Driven Development.

Here we create our tests before we write our code.

1. Doc Test:

Something that is unique to Python.

We write a doc string just when a function starts to say what the function does.

The tests are written similar to how we right code in a python shell, using three chevron symbol.

— Running doctest:

python -m dockets test.py

-m tells python to load the doc test module. doc test module runs the doc tests written in the test.py.

If nothing comes back on running the file, then its good. noting was failed.

Disadvantage:

can’t be reused as they are bound to the code they are written on.

all ofthere comparison is done through string comparison, so it can be tricky to compare floats etc.

2. UnitTest:

- Testing one particular aspect (method, function, etc) is called unit Testing.

- Its a package in Python to create unit tests for our tests.

- The methods in the unitest library should start with ‘test’ else they won’t run.

- We normally import the ‘unittest’ module and create a class that extends ‘unittest.TestCase’ class

Sample Code:

import unittest

class SomeTest(unittest.TestCase):

def setUp(self):

# adding some initial set up.

def test\_five\_plus\_five(self):

# The name of the method should start with test.

assert 5+5 = 10

def test\_one\_plus\_one(self):

assert not 1+1 =3

if \_\_name\_\_ == ‘\_\_main\_\_’:

unittest.main()

We can run this code by:

python -m unittest file\_name.py

python file\_name.py

——————————

- All testing libraries work on the concept of assertions.

- An assertion tests a condition in our code that must be met.

Quantitive Assertions:

a) assertEqual(a, b) - passes if a and b are equal.

b) assertNotEqual(a, b) - passes if a and b are not equal.

c) assertGreater(a, b) - passes if a is greater than b

d) assertGreaterEqual(a,b)

e) assertLessEqual(a, b)

f) assertLess(a, b) - passes if a is less than b

g) assertIn()

h) assertNotIn()

i) assertIsInstance(thisThing, isAnInstnaceofThisClass)

- Testing that our exceptions get raised:

assertRaises Assertion is used in this case.

How to use assertRaises:

- Its used a bit differently when compared to other assertions:

with self.assertRaises(ValueError):

# the type of exception that we need to confirm will be raised.

someMethod()

This test will pass only if the exception that we want is raised!.

Similar to ‘assertRaises’ we have ‘assertWarns’ and ‘assertLogs’ assertions. We can use these to check if warning or Log entries are created by our code.

Using Coverage:

This is library in Python that is used to tell if we have covered every aspect of our code in a test.

Install it using : pip install coverage

‘coverage run file\_name.py’ will simply run a tests

‘coverage report’ will give an overall report of % of areas that we have covered.

‘coverage report -m’ gives us the line of codes that we missed to include in our test.

‘coverage html’ will give a simple html version of our coverage.

​

Rules for Writing Python.

PEP 8 :

Python Enhancement Proposal

PEP 8 is Python's style guide. It's a set of rules for how to format your Python code to maximize its readability. Writing code to a specification helps to make large code bases, with lots of writers, more uniform and predictable, too.

Two blank lines between global function

We can run our file with ‘flake8’ to see if there are any errors related to PEP 8 convention.

DocString

A string at the beginning of the class or function that gives into about that method or class.

These very doc string are what is displayed in the help method of the shell..

import docstrings

help(docstrings.somemethod)

>> This will give us the help text.

Logging:

import logging

logging.basicConfig(filename='cc.log', level=logging.DEBUG)

logging.debug("DEBUGGED!")

The python Debugger!:

import pub

pdb.set\_trace() sets up the stoppage at a line.

We can print ’n’ or ‘next’ for next line of code

We can print ‘c’ or ‘continue’ to continue the execution

Remove the pdb line after debugging. So the most basic use of the this is inside the code as “

import pdb; pub.set\_trace() (the only place where we have semicolons in python)

(204, 255, 255)

​

Database

Important terms:

• model - A code object that represents a database table

• SqliteDatabase - The class from Peewee that lets us connect to an SQLite database

• Model - The Peewee class that we extend to make a model

• CharField - A Peewee field that holds onto characters. It's a varchar in SQL terms

• max\_length - The maximum number of characters in a CharField

• IntegerField - A Peewee field that holds an integer

• default - A default value for the field if one isn't provided

• unique - Whether the value in the field can be repeated in the table

• .connect() - A database method that connects to the database

• .create\_tables() - A database method to create the tables for the specified models.

• safe - Whether or not to throw errors if the table(s) you're attempting to create already exist

from peewee import \*

• TextField() - a field that holds a blob of text of any size

• DateTimeField() - a field for holding a date and a time

useful Methods:

.create() - creates a new instance all at once

.select() - finds records in a table

.save() - updates an existing row in the database

.get() - finds a single record in a table

.delete\_instance() - deletes a single record from the table

.order\_by() - specify how to sort the records

if \_\_name\_\_ == '\_\_main\_\_' - a common pattern for making code only run with the script is run and not when it's imported

db.close() - not a method we used, but often a good idea. Explicitly closes the connection to the database.

.update() - also something we didn't use. Offers a way to update a record without .get() and .save(). Example: Student.update(points=student['points']).where(Student.username == student['username']).execute()

db = SqliteDatabase("challenges.db")

class Challenge(Model):

name = CharField(max\_length=100)

language = CharField(max\_length=100)

class Meta:

database = db