**Python Date Type**

Python version 2 is different than version 3. This is only for version 3

**Python is not backward capable**

1. type int. integer has no specific byte that means it can hold up to any number <class 'int'>

#Literals

a= 12 # decimal Integer literal

a=0b101 # Binary Integer literal

a=Oo17   # octal Integer literal

a=Ox1AF # Hex Integer literal

a= up to any length

 #All of the above are Integer objects

2. type float 8-byte max <class 'float'>

a=12.5 #simple notation

a=1.2545644698e308 #Scientific notation

a=2.5e309 => inf

only 15 digits after decimal will be counted any value after that will be ignored. e308 is the highest possible value after that will be result in const inf(infinity)

3. complex

a= 2 +3j #complex -a+bj where a and b can be both int or float

print(type(a)) #<class ‘complex’>

print(a) #(2+3j)

4. bool (boolean)

a= True

b= False

print(type(a)) #<class ‘bool’>

print(a) #True

5.None

Last type is None type which means that value is null (not pointing to any objects).

a= None

print(type(a)) #<class ‘NoneType’>

print(a) #None

**More on Lists**

|  |  |
| --- | --- |
| list.append(*x*) | Add an item to the end of the list. Equivalent to a[len(a):] = [x]. |
| list.extend(*iterable*) | Extend the list by appending all the items from the iterable. Equivalent to a[len(a):] = iterable. |
| list.insert(*i*, *x*) | Insert an item at a given position. The first argument is the index of the element before which to insert, so a.insert(0, x) inserts at the front of the list, and a.insert(len(a), x) is equivalent to a.append(x). |
| list.remove(*x*) | Remove the first item from the list whose value is equal to x. It raises a [ValueError](https://docs.python.org/3/library/exceptions.html#ValueError) if there is no such item. |
| list.pop([*i*]) | Remove the item at the given position in the list, and return it. If no index is specified, a.pop() removes and returns the last item in the list. (The square brackets around the i in the method signature denote that the parameter is optional, not that you should type square brackets at that position. You will see this notation frequently in the Python Library Reference.) |
| list.clear() | Remove all items from the list. Equivalent to del a[:]. |
| list.index(*x*[, *start*[, *end*]]) | Return zero-based index in the list of the first item whose value is equal to x. Raises a [ValueError](https://docs.python.org/3/library/exceptions.html#ValueError) if there is no such item. The optional arguments start and end are interpreted as in the slice notation and are used to limit the search to a particular subsequence of the list. The returned index is computed relative to the beginning of the full sequence rather than the start argument. |
| list.count(*x*) | Return the number of times x appears in the list. |
| list.sort(*\**, *key=None*, *reverse=False*) | Sort the items of the list in place (the arguments can be used for sort customization, see [sorted()](https://docs.python.org/3/library/functions.html#sorted) for their explanation). |
| list.copy() | Return a shallow copy of the list. Equivalent to a[:]. |

**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 (“last-in, first-out”). 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.

**Using Lists as Queues**

It is also possible to use a list as a queue, where the first element added is the first element retrieved (“first-in, first-out”); however, lists are not efficient for this purpose. While appends and pops from the end of list are fast, doing inserts or pops from the beginning of a list is slow (because all of the other elements have to be shifted by one).

To implement a queue, use collections.deque which was designed to have fast appends and pops from both ends.

**Bytes**

#Internally is uses byte array structure

#byte is immutable sequence of byte in Key

#It dose allows duplicate entry

#It is ordered

Uses for serialization

**ByteArray**

#Ordered, allows duplicate values, mutable

Bytes and bytearray objects contain single bytes – the former is immutable while the latter is a mutable sequence. Bytes objects can be constructed the constructor, bytes(), and from literals; use a b prefix with normal string syntax: b'python'. To construct byte arrays, use the bytearray() function.

**Str- string**

#It is an immutable sequence of characters (if we change a new memory will be allocated)

#It maintains the order of characters

#It is sorted in a backup array internally

#for multi-line string use ‘’’ quotes also used for multi-line comments

#r prefix in string used to make special character meaning less

str1 = r'''mohammad \n loves python'''

print(str1) # r'''mohammad \n loves python

There are two types of for loop (iterable, range base)

Range: is built-in function

Use case is to generate a series of integer

R = range(5) # range(start, stop, step) or range(stop)

R1= range (5,0,-1)

**Range is a data type = everything is an object in Python**

r= range(5)

print(type(r)) #<class ‘range’>

print(r) #(0,5)

list1= list(r)

print(list1) #[0,1,2,3,4]

**Arithmetic operator**

(+, -, \* (Repetition), / ((int/int)-> int), // floor division, %Remainder, \*\* exponential)

is: operator comparing the memory address. If two object point to same memory address return true otherwise false.

==: is used to check equality in values (string and numbers)

List1= [3,4,5]

List1=list1\*3 #[3,4,5, 3,4,5, 3,4,5]

l= "mohammad"

l = l\*3

print(l) # mohammadmohammadmohammad

## Looping Techniques

When looping through dictionaries, the key and corresponding value can be retrieved at the same time using the items() method.

>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}

>>> for k, v in knights.items():

... print(k, v)

...

gallahad the pure

robin the brave

When looping through a sequence, the position index and corresponding value can be retrieved at the same time using the enumerate() function.

>>> for i, v in enumerate(['tic', 'tac', 'toe']):

... print(i, v)

...

0 tic

1 tac

2 toe

To loop over two or more sequences at the same time, the entries can be paired with the zip() function.

>>> questions = ['name', 'quest', 'favorite color']

>>> answers = ['lancelot', 'the holy grail', 'blue']

>>> for q, a in zip(questions, answers):

... print('What is your {0}? It is {1}.'.format(q, a))

...

What is your name? It is lancelot.

What is your quest? It is the holy grail.

What is your favorite color? It is blue.

To loop over a sequence in reverse, first specify the sequence in a forward direction and then call the reversed() function.

>>> for i in reversed(range(1, 10, 2)):

... print(i)

...

9

7

5

3

1

To loop over a sequence in sorted order, use the sorted() function which returns a new sorted list while leaving the source unaltered.

>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']

>>> for i in sorted(basket):

... print(i)

...

apple

apple

banana

orange

orange

pear

Using set() on a sequence eliminates duplicate elements. The use of sorted() in combination with set() over a sequence is an idiomatic way to loop over unique elements of the sequence in sorted order.

>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']

>>> for f in sorted(set(basket)):

... print(f)

...

apple

banana

orange

pear

It is sometimes tempting to change a list while you are looping over it; however, it is often simpler and safer to create a new list instead.

>>> import math

>>> raw\_data = [56.2, float('NaN'), 51.7, 55.3, 52.5, float('NaN'), 47.8]

>>> filtered\_data = []

>>> for value in raw\_data:

... if not math.isnan(value):

... filtered\_data.append(value)

...

>>> filtered\_data

[56.2, 51.7, 55.3, 52.5, 47.8]

## More on Conditions

The conditions used in while and if statements can contain any operators, not just comparisons.

The comparison operators in and not in are membership tests that determine whether a value is in (or not in) a container. The operators is and is not compare whether two objects are really the same object. All comparison operators have the same priority, which is lower than that of all numerical operators.

Comparisons can be chained. For example, a < b == c tests whether a is less than b and moreover b equals c.

Comparisons may be combined using the Boolean operators and and or, and the outcome of a comparison (or of any other Boolean expression) may be negated with not. These have lower priorities than comparison operators; between them, not has the highest priority and or the lowest, so that A and not B or C is equivalent to (A and (not B)) or C. As always, parentheses can be used to express the desired composition.

The Boolean operators and and or are so-called *short-circuit* operators: their arguments are evaluated from left to right, and evaluation stops as soon as the outcome is determined. For example, if A and C are true but B is false, A and B and C does not evaluate the expression C. When used as a general value and not as a Boolean, the return value of a short-circuit operator is the last evaluated argument.

It is possible to assign the result of a comparison or other Boolean expression to a variable. For example,

str1 , str2, str3 = '','mohammad','aziz'

non\_null = str1 or str2 or str3

print(non\_null)#mohammad

Note that in Python, unlike C, assignment inside expressions must be done explicitly with the walrus operator :=. This avoids a common class of problems encountered in C programs: typing = in an expression when == was intended.

## [Why can’t I use an assignment in an expression?](https://docs.python.org/3/faq/design.html#id8)

Starting in Python 3.8, you can!

Assignment expressions using the walrus operator := assign a variable in an expression:

while chunk := fp.read(200):

print(chunk)

See [**PEP 572**](https://www.python.org/dev/peps/pep-0572) for more information.

## Comparing Sequences and Other Types

Sequence objects typically may be compared to other objects with the same sequence type. The comparison uses lexicographical ordering: first the first two items are compared, and if they differ this determines the outcome of the comparison; if they are equal, the next two items are compared, and so on, until either sequence is exhausted. If two items to be compared are themselves sequences of the same type, the lexicographical comparison is carried out recursively. If all items of two sequences compare equal, the sequences are considered equal. If one sequence is an initial sub-sequence of the other, the shorter sequence is the smaller (lesser) one. Lexicographical ordering for strings uses the Unicode code point number to order individual characters. Some examples of comparisons between sequences of the same type:

(1, 2, 3) < (1, 2, 4)

[1, 2, 3] < [1, 2, 4]

'ABC' < 'C' < 'Pascal' < 'Python'

(1, 2, 3, 4) < (1, 2, 4)

(1, 2) < (1, 2, -1)

(1, 2, 3) == (1.0, 2.0, 3.0)

(1, 2, ('aa', 'ab')) < (1, 2, ('abc', 'a'), 4)

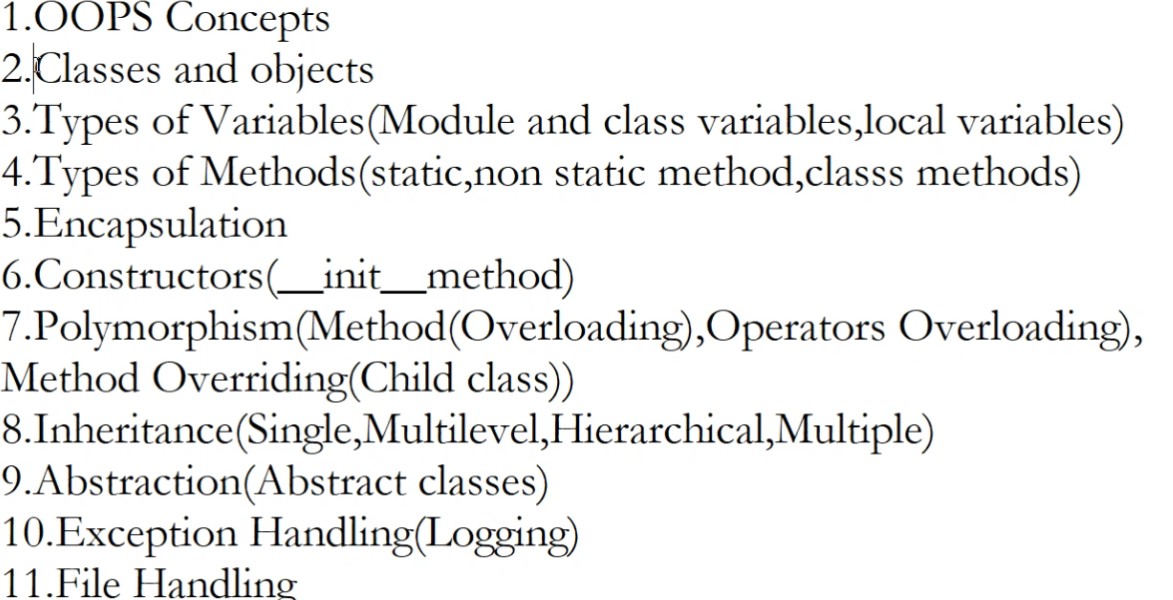
Note that comparing objects of different types with < or > is legal provided that the objects have appropriate comparison methods. For example, mixed numeric types are compared according to their numeric value, so 0 equals 0.0, etc. Otherwise, rather than providing an arbitrary ordering, the interpreter will raise a [TypeError](https://docs.python.org/3/library/exceptions.html#TypeError) exception.

**Python Intermediate**

1. OOPs Concepts
2. Classes and objects
3. Types of Variables (Module and class variable, local variable)
4. Types of Methods
5. Encapsulation
6. Construction (\_\_init\_\_ method)
7. Polymorphism (Method Overloading, Operator Overloading,)

Method Overriding

1. Inheritance (single, Multilevel, Hierarchical, Multiple)
2. Abstraction (Abstract class)
3. Exception Handling (Logging)
4. File Handling
5. Regular Expression
6. Lambda function, generators, iterators



Since there is default argument python will not support method overloading

**Benefits of OOP**

So why use OOP? Why not use some other paradigm? To be clear, OOP isn't better or worse than any other paradigm. There are pros and cons to everything. OOP does have some nice benefits, and here are some of those benefits:

* **Data encapsulation**: Data encapsulation is about hiding data away from the rest of the system and only allowing access to parts of it. The reason is data holds *state*, and that state can be made up of one or more variables. If these variables need to be changed at the same time, you need to protect them and only allow access via public methods so that changes are made in a predictable way. OOP has mechanisms like access levels, where data that's on an object can only be accessed by the object itself or can be made publicly available.
* **Simplicity**: Building large systems is a complex task, with many problems to solve. Being able to break down the complexity into smaller problems, to objects, means you can *simplify* the overall task.
* **Easy to modify**: When you rely on objects and model your system with them, it's easier to track down what parts of the system need modifying. For example, you might need to correct a bug or add a new feature.
* **Maintainability**: Maintaining code in general is hard, and it becomes harder over time. It requires discipline in the form of good naming and a clear and consistent architecture, among other things. Using objects makes it easier to locate a specific area of your code that needs maintaining.

**Reusability**: An object's definition can be used many times in many parts of your system or potentially in other systems too. When you reuse code, you save time and money because you need to write less code and you reach your target faster

In Python, the constructor has the name \_\_init()\_\_. You also need to pass a special keyword, self, as a parameter to the constructor. The keyword self refers to the object's instance.

**Access Modifiers in Python:** Public, Private and Protected

A Class in Python has three types of access modifiers:

* **Public Access Modifier**
* **Protected Access Modifier**
* **Private Access Modifier**

## Public Access Modifier:

The members of a class that are declared public are easily accessible from any part of the program. All data members and member functions of a class are public by default.

# program to illustrate public access modifier in a class

class Geek:

# constructor

def \_\_init\_\_(self, name, age):

# public data members

self.geekName = name

self.geekAge = age

# public member function

def displayAge(self):

# accessing public data member

print("Age: ", self.geekAge)

# creating object of the class

obj = Geek("R2J", 20)

# accessing public data member

print("Name: ", obj.geekName)

# calling public member function of the class

obj.displayAge()

In the above program, geekName and geekAge are public data members and displayAge() method is a public member function of the class Geek. These data members of the class Geek can be accessed from anywhere in the program.

## Protected Access Modifier:

The members of a class that are declared protected are only accessible to a class derived from it. Data members of a class are declared protected by adding a single underscore ‘\_’ symbol before the data member of that class.

# program to illustrate protected access modifier in a class

# super class

class Student:

     # protected data members

     \_name = None

     \_roll = None

     \_branch = None

     # constructor

     def \_\_init\_\_(self, name, roll, branch):

          self.\_name = name

          self.\_roll = roll

          self.\_branch = branch

     # protected member function

     def \_displayRollAndBranch(self):

          # accessing protected data members

          print("Roll: ", self.\_roll)

          print("Branch: ", self.\_branch)

# derived class

class Geek(Student):

       # constructor

       def \_\_init\_\_(self, name, roll, branch):

                Student.\_\_init\_\_(self, name, roll, branch)

       # public member function

       def displayDetails(self):

                 # accessing protected data members of super class

                print("Name: ", self.\_name)

                 # accessing protected member functions of super class

                self.\_displayRollAndBranch()

# creating objects of the derived class

obj = Geek("R2J", 1706256, "Information Technology")

# calling public member functions of the class

obj.displayDetails()

## Private Access Modifier:

The members of a class that are declared private are accessible within the class only, private access modifier is the most secure access modifier. Data members of a class are declared private by adding a double underscore ‘\_\_’ symbol before the data member of that class.

# program to illustrate private access modifier in a class

class Geek:

     # private members

     \_\_name = None

     \_\_roll = None

     \_\_branch = None

     # constructor

     def \_\_init\_\_(self, name, roll, branch):

          self.\_\_name = name

          self.\_\_roll = roll

          self.\_\_branch = branch

     # private member function

     def \_\_displayDetails(self):

           # accessing private data members

           print("Name: ", self.\_\_name)

           print("Roll: ", self.\_\_roll)

           print("Branch: ", self.\_\_branch)

     # public member function

     def accessPrivateFunction(self):

           # accessing private member function

           self.\_\_displayDetails()

# creating object

obj = Geek("R2J", 1706256, "Information Technology")

# calling public member function of the class

obj.accessPrivateFunction()

In the above program, \_\_name, \_\_roll and \_\_branch are private members, \_\_displayDetails() method is a private member function (these can only be accessed within the class) and accessPrivateFunction() method is a public member function of the class Geek which can be accessed from anywhere within the program. The accessPrivateFunction() method accesses the private members of the class Geek.

**Use decorators for getters and setters**

Decorators are an important subject in Python. They're part of a larger subject called meta programming. Decorators are functions that take your function as an input. The idea is to encode reusable functionality as decorator functions and then decorate other functions with it. The purpose is to give your function a feature it didn't have before. A decorator can, for example, add fields to your object, measure the time it takes to invocate a function, and do much more.

In the context of OOP and getters and setters, a specific decorator @property can help you remove some boilerplate code when you add getters and setters. The @property decorator does the following things for you:

* **Creates a backing field**: When you decorate a function with the @property decorator, it creates a backing private field. You can override this behavior if you want, but it's nice to have a default behavior.
* **Identifies a setter**: A setter method can change the backing field.
* **Identifies a getter**: This function should return the backing field.
* **Identifies a delete function**: This function can delete the field.

class Square:

def \_\_init\_\_(self, w, h):

self.height = h

self.\_\_width = w

def set\_side(self, new\_side):

self.\_\_height = new\_side

self.\_\_width = new\_side

@property

def height(self):

return self.\_\_height

@height.setter

def height(self, new\_value):

if new\_value >= 0:

self.\_\_height = new\_value

else:

raise Exception("Value must be larger than 0")

In the preceding code, the function height() is decorated by the decorator @property. This decoration action creates the private field \_\_height. The \_\_height field isn't defined in the constructor \_\_init\_\_() because the decorator does that already. There's also another decoration happening, namely, @height.setter. This decoration points out a similar-looking height() method as the setter. The new height method takes an another parameter value as its second parameter.

Being able to manipulate the height separate from the width will still cause a problem. You'll need to understand what the class does before you consider allowing getters and setters, because you're introducing risk.

Operator overlodaing