Contents

[1 Some Python related concepts / topics 2](#_Toc529730034)

[1.1 The Python Debugger 2](#_Toc529730035)

[1.2 Everything is an object (mutable or immutable) in Python 2](#_Toc529730036)

[1.2.1 The basics of variable assignments 2](#_Toc529730037)

[1.2.2 Mutability – Imuutability Table 3](#_Toc529730038)

[1.2.3 Id() and Type() functions 3](#_Toc529730039)

[1.2.4 Actual examples with code and diagrams 3](#_Toc529730040)

[1.2.5 So, is Python object oriented ? 7](#_Toc529730041)

[1.3 Indentation in Python 7](#_Toc529730042)

[1.3.1 Sample Images 7](#_Toc529730043)

[1.3.2 End of a statement, Blocks of Code (“suites”) 9](#_Toc529730044)

[1.4 Functions and modules 10](#_Toc529730045)

[1.4.1 In what order does Python look for modules 10](#_Toc529730046)

[1.5 Iteration in Python 10](#_Toc529730047)

[1.6 Data Structures 11](#_Toc529730048)

[1.6.1 General Write Up 11](#_Toc529730049)

[1.6.2 Some useful links related to data structures and algorithms in general 11](#_Toc529730050)

[1.6.3 Python Dictionaries 11](#_Toc529730051)

# Some Python related concepts / topics

## The Python Debugger

Serious programming will always involve debugging. You’ve earned your stripes only after you have debugged yours and code developed by other programmers.

In addition to locating hard to solve problems, it also helps you understand how a program works.

The module pdb defines an interactive source code debugger for Python and is very widely used. Link for detailed information - <https://docs.python.org/3/library/pdb.html>.

There are other debuggers as well. PyCharm, a popular Python IDE, comes with an integrated debugger.

## Everything is an object (mutable or immutable) in Python

### The basics of variable assignments

When learning Python, one has to quickly understand that is that all objects in Python are either **mutable** or **immutable**.

Everything in Python is an Object, every variable that we create holds an object instance.

* name = “pibm” : <name> = <object>
* Val = 1 : <name> = <object>
* myList = [1,2,3] : <name> = <object>

When an object is created *(Other terms used – “instantiated”, “initialized”)*, it is assigned a unique object id.

Its type is defined at runtime and once set can never change, however its state can be changed if it is mutable.

In short, the state (“values”) of a **mutable** object can be changed after it is created, and an **immutable** object’s state (“values”) can’t.

When we assign a value to a variable :

We are actually **binding** a **name** to an **object.**One implication of this is that multiple names can be bound to a single object. (Multiple labels can be assigned to the same object).

The variable is actually a label that we assign to an object, it is the way we, as developers can identify it. However, what is always important about the underlying object is its value (“state”) and its type.

### Mutability – Imuutability Table

|  |  |  |  |
| --- | --- | --- | --- |
| **Class** | **Description** | **Immutable** | **Remarks** |
| Bool | Boolean Value | YES |  |
| Int | Integers | YES |  |
| Float | Floating point numbers | YES |  |
| List | Mutable sequence of objects | NO |  |
| Tuple | Immutable sequence of objects | YES |  |
| Str | Character string | YES |  |
| Set | Unordered set of distinct objects | NO |  |
| Frozenset | Immutable form of “set” class | YES |  |
| Dict | Key value pairs  (associative mappings) | NO |  |

### Id() and Type() functions

* Id() - returns the actual memory location where the variable is stored
* type() – returns the type of the object that the variable is bound to

### Actual examples with code and diagrams

* Example 001 (immutable types)

|  |
| --- |
| #py\_everything\_is\_an\_obj\_001.py  a = "spam"  b = "spam"  #  print(id(a))  print(id(b))  #  # id() returns the actual memory location where the variable is stored.  # Since id(a) = id(b), we know that a and b both point to a single variable,  # that resides in a single memory location.  # This is what we mean by “multiple names bound to single object  print(a is b) |

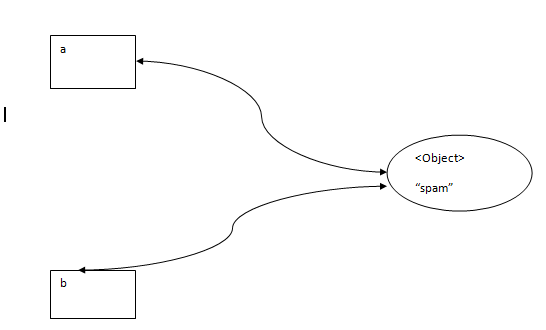
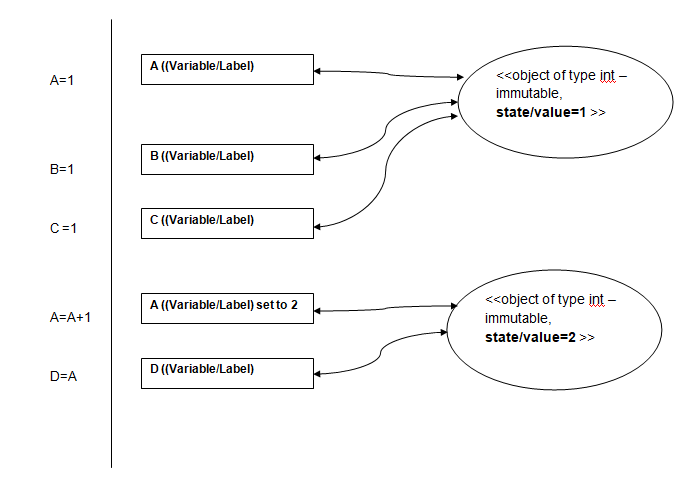
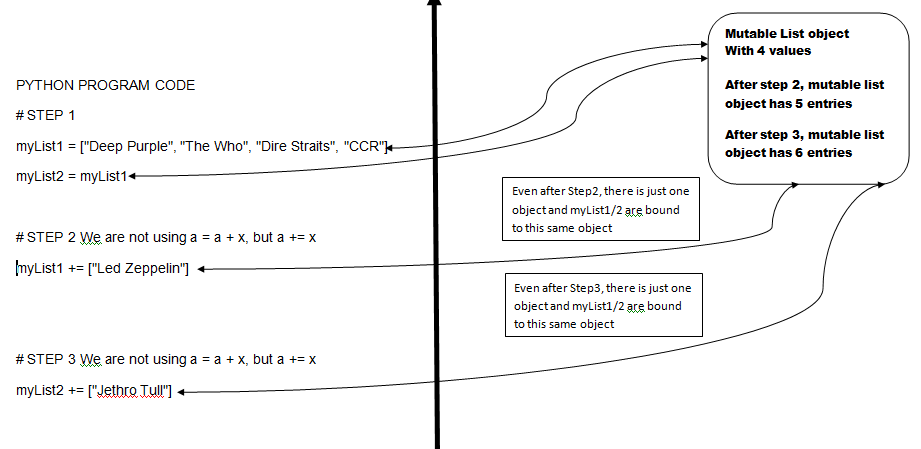


Figure – two variables bound to the same immutable object

* Example 002 (immutable types)



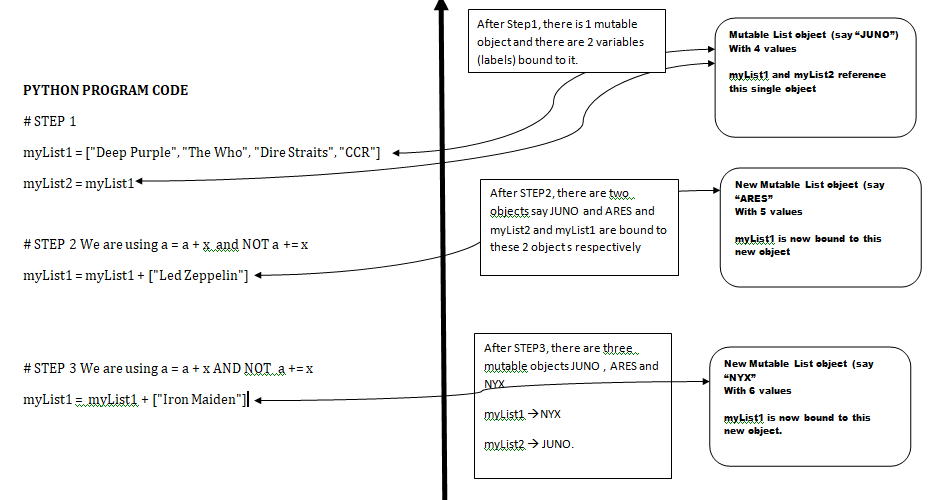
* Example 003 (with mutable types – LIST [] )



* Example 004 (with mutable types – LIST [] )

*Please compare this with Example 003. There is a big difference in behavior when using a = a +x and a+=x.*

*[ https://stackoverflow.com/questions/15376509/when-is-i-x-different-from-i-i-x-in-python ]*



### So, is Python object oriented ?

Everything in Python behaves as if it is an object derived from some class. In this way, you can think of Python as being more object based as opposed to purely object-oriented.

As everything is an object in Python, any “thing” can be assigned to any variable, and variables can be assigned *anything* (regardless of what the thing is: a number, a string, a function, a widget...any object).

**More on this later ….**

## Indentation in Python

First, let us go through some images which are snippets of code

### Sample Images

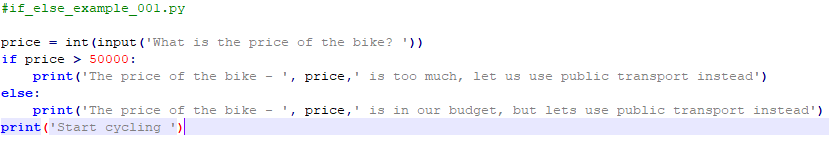


Figure – This has a simple if else construct

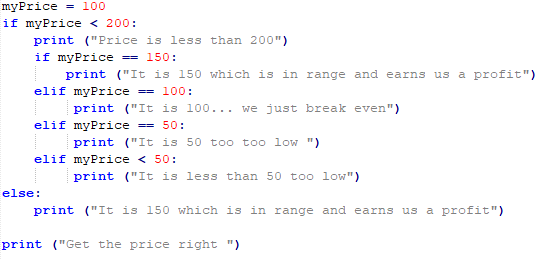


Figure – if elif within an if else. Nesting

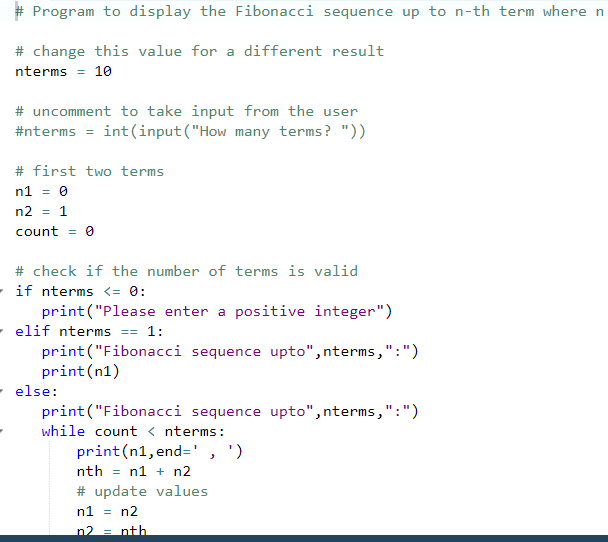


Figure – if –elif and while. Nesting

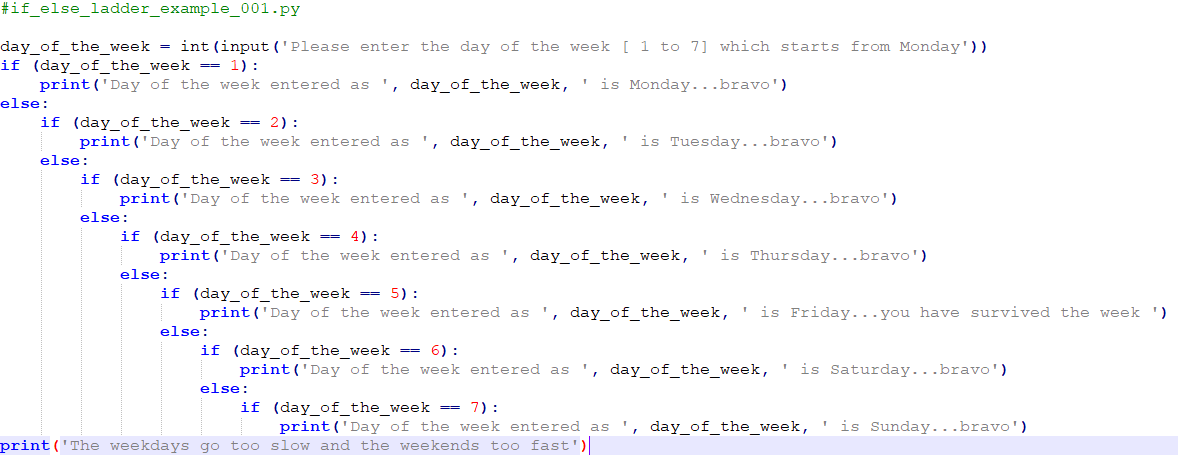


Figure – multiple levels of nesting (this kind of code is “code from hell”)

### End of a statement, Blocks of Code (“suites”)

|  |  |  |
| --- | --- | --- |
| * What is the end of a statement ? | you do not have to type in a semicolon or other special character; you simply press Enter |  |
| * What constitutes a block of code ? | To indicate a block of code in Python, you must indent each line/statement of the block by the same amount. |  |
| * How many spaces should you indent ? | Be consistent  Developers generally use 4 spaces  Whichever indent distance you choose, you must use this same distance consistently throughout a Python program |  |
| * Should you mix TABS and spaces ? | NEVER.  A special place in HELL awaits you if you do this !.  By the way, Python 3 will not let you mix spaces and tabs in the same file |  |

## Functions and modules

### In what order does Python look for modules

In most cases, the order is as follows :-

* Your current working directory
* Your interpreter’s site-packages locations
* The standard library locations

## Exception handling in Python

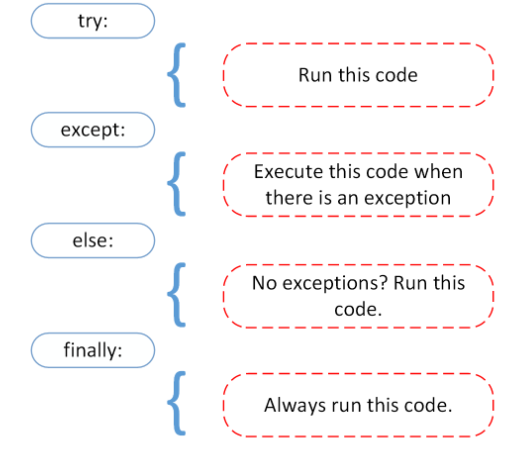


Figure : High level code structure for exception handling

### In what order does Python look for modules

In most cases, the order is as follows :-

* Your current working directory
* Your interpreter’s site-packages locations
* The standard library locations

## Iteration in Python

Repeated execution of a set of statements is called **iteration**

Because iteration is so common, Python provides several language features to make it easier.

We’ve already seen a bit of this in the sample programs.

The “for” loop. This is the form of iteration, you’ll likely be using most often. But in this chapter we’ve going to look at the “while” statement — another way to have your program do iteration, useful in slightly different circumstances.

Running through all the items in a list is called **traversing** the list, or **traversal**. One more term to absorb.

Choosing between for and while

Use a for loop if you know, before you start looping, the maximum number of times that you’ll need to execute the body. For example, if you’re traversing a list of elements, you know that the maximum number of loop iterations you can possibly need is “all the elements in the list”. Or if you need to print the 12 times table, we know right away how many times the loop will need to run.

So any problem like “iterate this weather model for 1000 cycles”, or “search this list of words”, “find all prime numbers up to 10000” suggest that a for loop is best.

In contrast, if you are required to repeat some computation until some condition is met, and you cannot calculate in advance when (of if) this will happen, you’ll need a while loop.

We call the first case **definite iteration** — we know ahead of time some definite bounds for what is needed. The latter case is called **indefinite iteration** — we’re not sure how many iterations we’ll need — we cannot even establish an upper bound!

## Data Structures

### General Write Up

Data structures are a way of organizing and storing data so that this data can be accessed and operated upon in an efficient manner.

As far as possible, the data should be stored and analyzed in memory for performance reasons.

Today programming languages like Python have various kinds of data structures defined that make it easier for data scientists and computer engineers, alike to concentrate on the main picture of solving larger problems rather than getting lost in the details of data description and access.

### Some useful links related to data structures and algorithms in general

<http://interactivepython.org/runestone/static/pythonds/index.html>

### Python 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" \l "ValueError" \o "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" \l "ValueError" \o "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.**reverse**()  Reverse the elements of the list in place.  list.**copy**()  Return a shallow copy of the list. Equivalent to a[:]. |

### Python Dictionaries