Contents

[1 Some Python related concepts / topics 2](#_Toc528997702)

[1.1 The Python Debugger 2](#_Toc528997703)

[1.2 Everything is an object (mutable or immutable) in Python 2](#_Toc528997704)

[1.2.1 The basics of variable assignments 2](#_Toc528997705)

[1.2.2 Mutability – Imuutability Table 3](#_Toc528997706)

[1.2.3 Id() and Type() functions 3](#_Toc528997707)

[1.2.4 Actual examples with code and diagrams 3](#_Toc528997708)

[1.3 Indentation in Python 4](#_Toc528997709)

[1.4 Iteration in Python 6](#_Toc528997710)

# Some Python related concepts / topics

## The Python 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** |  |
| 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

|  |
| --- |
| #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) |

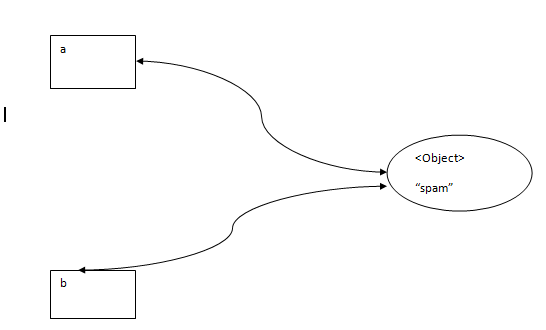
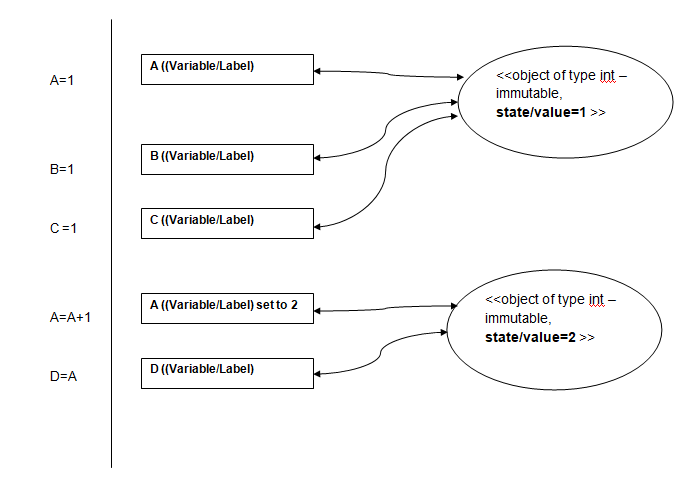


Figure –

* Example 002



## Indentation in Python

First, let us go through some images

### Sample Images

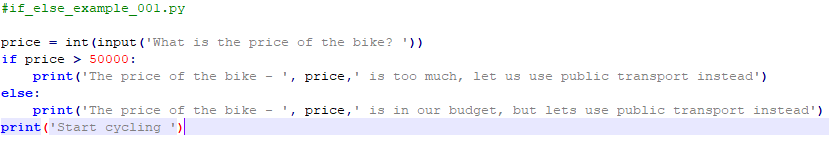


Figure – This has a simple if else construct

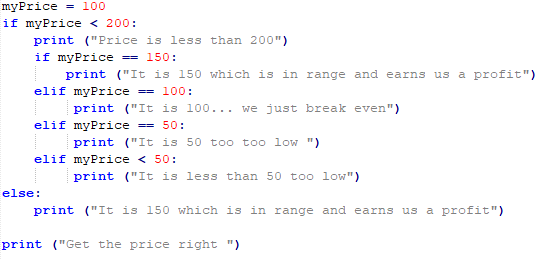


Figure – if elif within an if else. Nesting

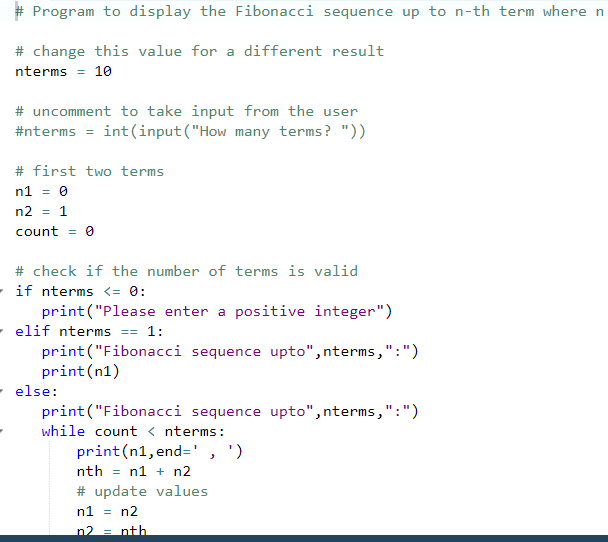


Figure – if –elif and while. Nesting

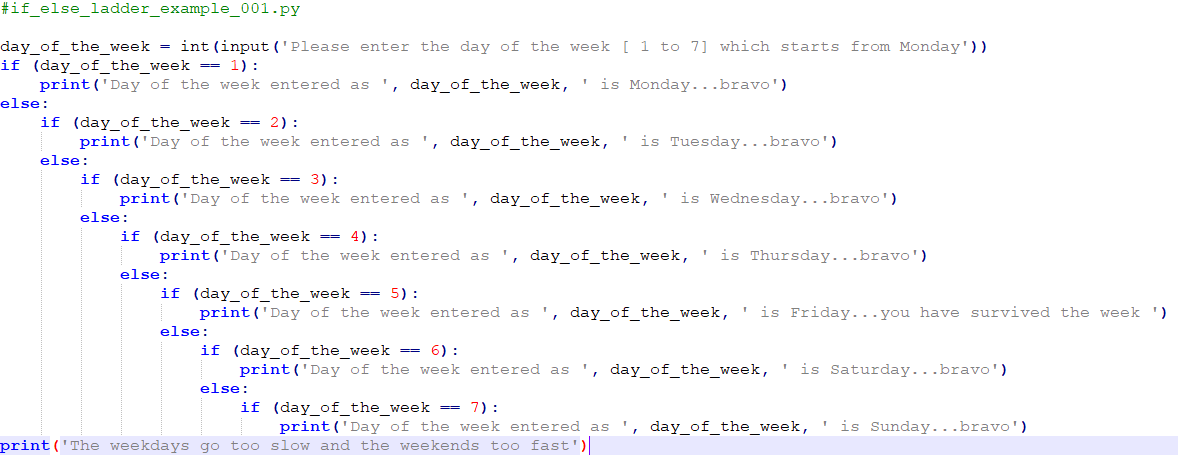


Figure – multiple levels of nesting (this kind of code is “code from hell”)

### 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  Hell awaits you if you do this  BTW, Python 3 will not let you mix spaces and tabs in the same file |  |

## 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 the for statement in chapter 3. This the 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**.

**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.

By 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, as we did in this 3n + 1 problem, 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!