**Website**: http://dcsqrd.com
**Email**: contact@dcsqrd.com

# Python 3

**(Intro excerpted from Python for Informatics)**

## 1. Introduction

Programming is a very creative and rewarding activity. You can write programs for many reasons, ranging from making your living to solving a difficult data analysis problem to having fun to helping someone else solve a problem. We believe that everyone needs to know how to program, and that once you know how to program you will figure out what you want to do with your newfound skills.

There are many things that you might need to do, that you could offload to a computer. If you know the language, you can "tell" the computer to do tasks that were repetitive. Interestingly, the kinds of things computers can do best are often the kinds of things that we humans find boring and mind-numbing. For example, you can easily read and understand the above paragraph, but if I ask you to tell me the word that is used most, counting them is almost painful because it is not the kind of problem that human minds are designed to solve.

This very fact that computers are good at things that humans are not is why you need to become skilled at talking "computer language". Once you learn this new language, you can delegate mundane tasks to your computer, leaving more time for you to do the things that you are uniquely suited for. You bring creativity, intuition, and inventiveness to this partnership.

You will need two skills to be a programmer:

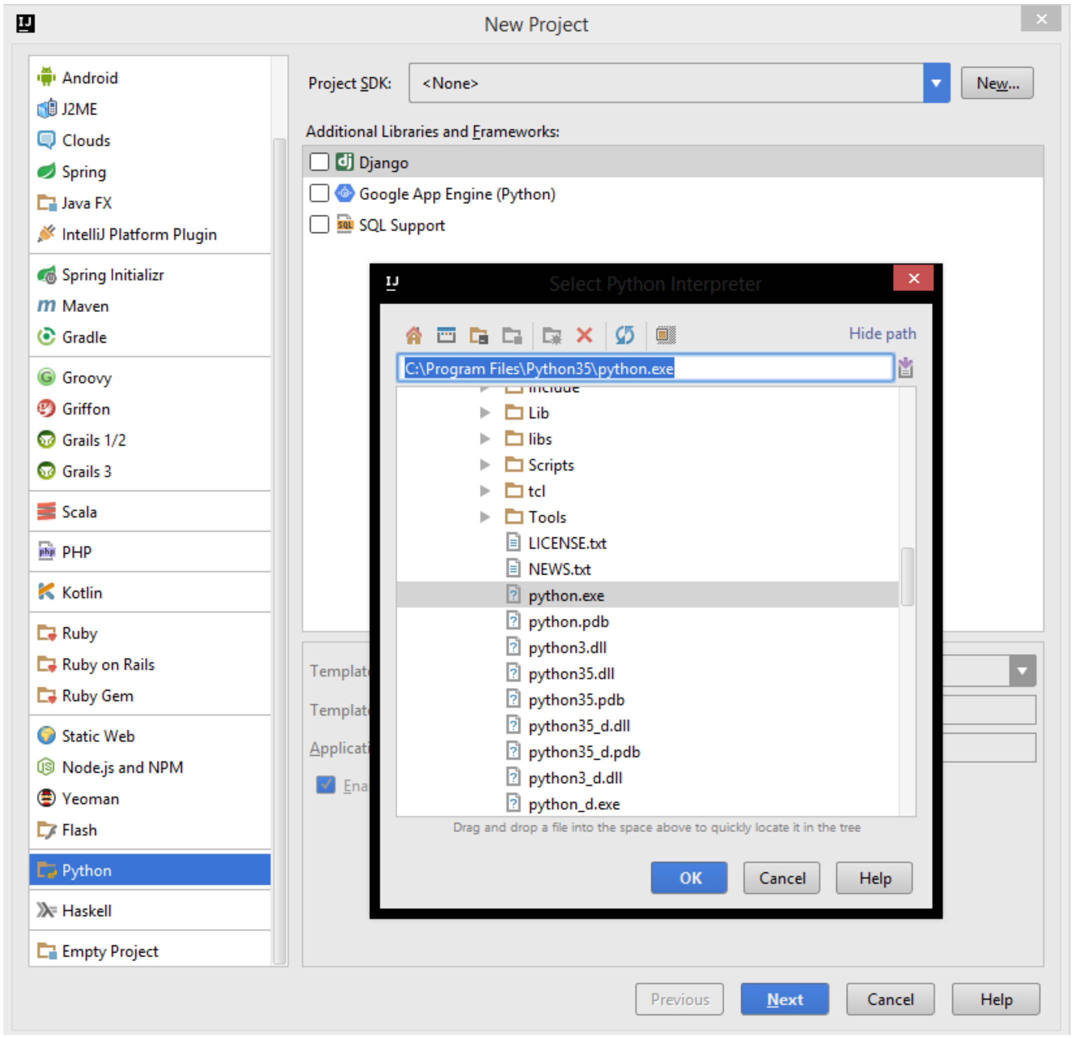
1. (The easy one) You need to know the vocabulary and grammar to spell out the "words" in this new language, so the computer understands
2. (The hard one) You need to combine these words and sentences to convey an idea, tell a story to the reader. The "Story" is the problem you are trying to solve, the "Idea" is the solution.

Once you get the hang of seeing patterns in problems, programming becomes a very pleasant and creative process.

## 2. Installing Python

This section will tell you where to download and install the Python IDE (IDLE)

- https://www.python.org/downloads/
- Download and install like any other program, no license required, completely free!
- Remember, Python 3 and Python 2 are not compatible! Download 3 for this workshop.
- For a better IDE, download PyCharm Community from https://www.jetbrains.com/pycharm/download/
- During initial configuration, you will need to point PyCharm to where Python (select Local) is installed. This is the Project SDK.



Project SDK: <None>

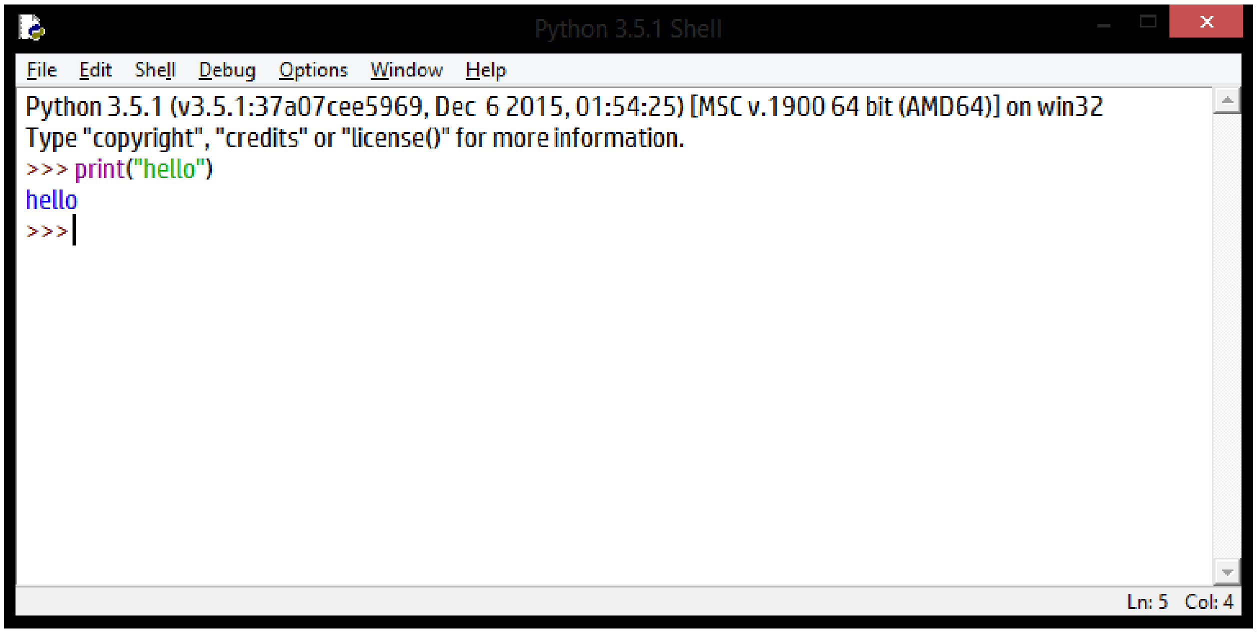
C:\Program Files\Python35\python.exe

## 3. Differences between Python and C++

| **Python** | **C/C++, Java** |
|---|---|
| Interpreted line by line | Compiled in one go |
| Dynamically typed | Strictly typed |
| Large number of libraries | Fewer standard libraries |
| Follows natural language principles | More symbol-oriented |
| Indentation-based | Space-insensitive |
| Scripting language | Procedure-based programming |
| No pointer support, only references | Pointers for direct memory manipulation |

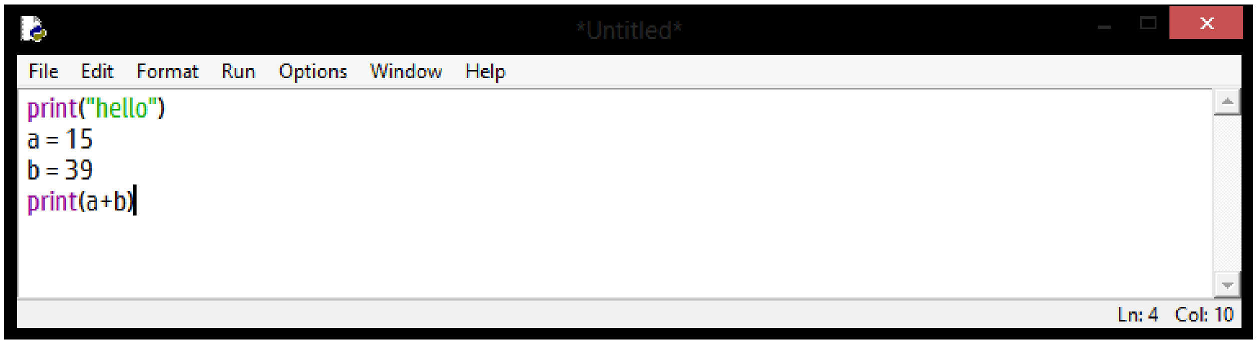
## 4. Conversing with Python

- Open IDLE, let's start from there. The >>> is called the "Interactive Chevron Prompt"



```
Python 3.5.1 (v3.5.1:37a07cee5969, Dec  6 2015, 01:54:25) [MSC v.1900 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> print("hello")
hello
>>>
```

- This is like the Linux terminal, you can "script" from here – mainly one-line commands. To write a "batch" of commands, press `ctrl`+`n`, to open a new Python file.



```
print("hello")
a = 15
b = 39
print(a+b)
```

- Type the program, save it with `ctrl`+`s`, press `f5` to run. Output will appear in the console window.

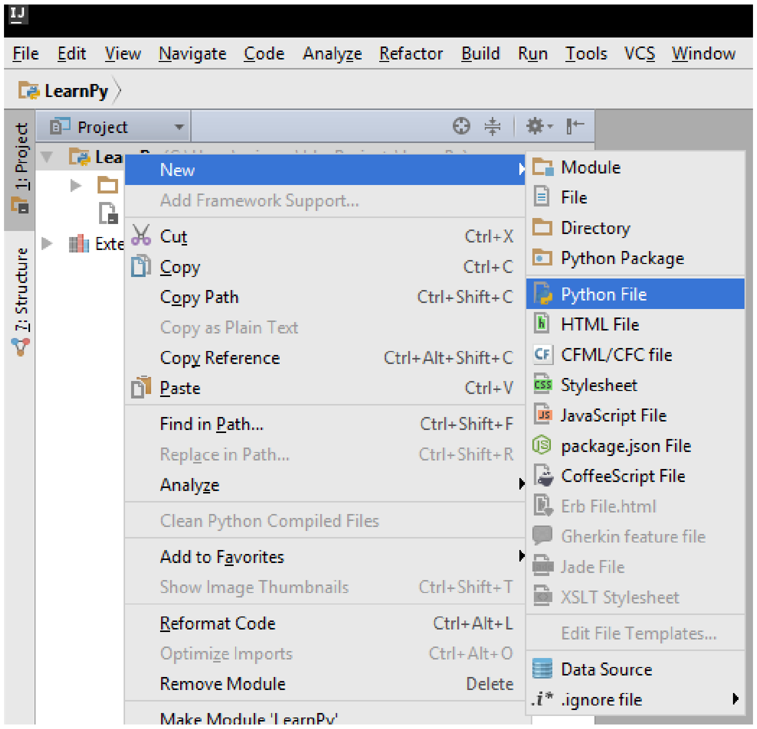
## 5. Your first program

Python is interpreted and not compiled, meaning it "understands" the program one line at a time. Python's philosophy is to keep things as simple as possible, so there is not a lot of boilerplate code.

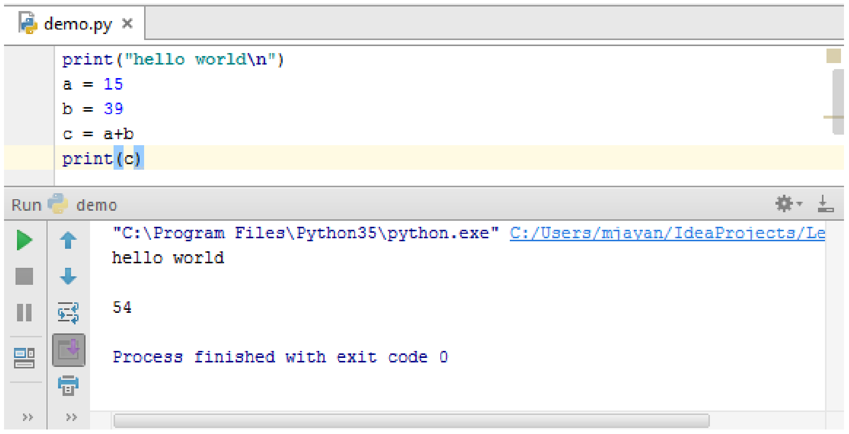
To print "hello world!" just type **`print("hello world!")`** into the chevron prompt or a new file.

## 6. A more intelligent IDE

- If you have PyCharm installed, open it. Let us try a program there. Create a new project, name it LearnPy.
- Create a new Python file from the context menu of the project.
- Enter your code into the file. PyCharm will suggest code with syntax and parameters where necessary.



- Run the code with `alt`+`shift`+`f10`, select which file to execute.
- Output appears in an integrated window.



```
print("hello world\n")
a = 15
b = 39
c = a+b
print(c)
```

```
"C:\Program Files\Python35\python.exe" C:/Users/mjayan/IdeaProjects/Le
hello world

54

Process finished with exit code 0
```

## 7. Data types in Python

There are only three default data types in Python. There is no differentiation like **`int`**, **`float`**, **`char`**, **`double`** or **`string`**.

### 7.1 Primitives

Primitives, as the name suggest, are primitive values bound to a variable. For example, **`a=15`** is a primitive assignment, where the variable name is **`a`** and the primitive value is **`15`**.

Similarly, floats, strings can be assigned to variables. Example:

```
str = “Good, you are learning Python!”
pi = 3.14159
```

You can reassign primitives dynamically. For example,

```
a=15
a=18.67
a = “Python is awesome!”
```

There is also a special set of primitives:

- **`True`** and **`False`**, which are Booleans. Note that they do not have quotes around them, to differentiate from strings.
- **`None`**, which indicates the absence of any value. Functions equivalent is the **`NotImplemented`** type. Neither are used much in interest of best practices.

## 7.2 Lists

Lists contain a set of values accessed by the "index". Index is a contiguous integer sequence that starts with 0. Example: **`arr=[10,20,35]`** means **`a[0]=10`**, **`a[1]=20`** and **`a[2]=35`**.

Note that, since Python does not differentiate between **`int`**, **`float`**, **`char`** and **`string`**s, it is possible to have mixed types of data inside a list, unlike C-like languages. For instance, **`arr=[10,'a',"Hello!\n",22.7895,"Learn Python!"]`** is absolutely valid. Accessing elements are similar with normal lists, and types can change dynamically. Python calls this a "list" to differentiate from arrays (in other languages), which usually contain elements of similar data types. Note that lists can contain lists as elements, like this:

```
['spam', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]] # length=4
```

The operator can be used to check if an element is in a given list. For example:

```
list = [3, 5, 7, 14]
3 in list               # returns True
'adam' in list          # returns False
```

Some notable functions on lists are **`append()`**, **`extend()`**, **`sort()`**, **`pop()`**, **`del()`**, **`remove()`**, **`max()`**, **`min()`**, **`sum()`**,

## 7.3 Dictionaries

Dictionaries are like hash maps or associative arrays in other languages. For the uninitiated, they are key-value pairs. Dictionaries are similar to lists, except elements are accessed via "key"s. Example,

```
hash = {"name":"John","age":21,"college":"NIE IT"}
```

Accessing elements is done like this:

**`hash["name"]`** is **`"John"`**, **`hash["age"]`** is **`21`**, and so on

Note that dictionaries are delimited by **`{}`** and elements are separated by **`,`** whereas keys and values are separated by **`:`**

Some important patterns for dictionaries are:

- ***`dictionary`*`.get(`*`key`*`,`*`default`*`)`**: returns the value if key is present in dictionary, default value otherwise.
- **`for`** ***`var`*** **`in`** ***`dictionary`***: returns the list of keys in the dictionary, *in no particular order.*

### 7.4 Tuples

Tuples are like lists, except they are immutable. Values can be of any type, and are comparable and hashable. Just like lists, they are indexed by integers. Tuples are enclosed by round brackets, although it is not strictly necessary. Example, **`t = ('a','b','c','d', 'e')`**. If you need to create a tuple with a single value, remember to use a comma after the first element, like **`t = (‘a’,)`** to differentiate from string assignments.

## 8. Reading input from and writing output to the console

- The **`input(`*`[prompt]`*`)`** method is used to take input from the terminal. It takes in a string prompt and returns the keyboard input as a string.
```
speed = input(“Enter your speed”)
```
- **`print()`** is used to print data to the console. Strings can be separated by commas or concatenation operator(**+**) in Python-style **`print()`**s, or **`%d`**, **`%f`** C-style prints can be used with **`%variable_name`** in posterior order.
```
print(“Language”+lang,“Version”,ver)      #Python style print()
print(“You are using %s %d”,%lang, %ver)  #C-style print()
```

## 9. Operators

Operators work pretty much like C-based languages, but with a few differences:

- Python supports exponentiation operation via the **`**`** operator
- Division operator **`/`** produces floating point values by default, but **`//`** performs integer division *(not applicable to Python 2.x)*
- Increment and decrement operators are not supported, use **`+=1`** or **`-=1`** instead
- **`+`** is used to concatenate strings, if one operand is a string, the rest are converted into strings by default. Be careful!
```
a=”2”
b = 3
print(a+b)     #This prints 23!
```
- **`*`** operator can be used to repeat a sequence. For example, **`3*’un’+’ium’`** produces **`ununnium`**
- All data input from the user (using **`input()`** function) are strings by default. You will need to convert it into numeric types explicitly when necessary.
- Use **`str()`** to convert any data to a string, and **`int()`** to convert data into integer (if supported). Similarly, **`float()`**, **`chr()`** can be used to construct these types.
- **`==`** and **`!=`** can be used as usual in Python, but there is a stronger set of equality and non-equality comparison operators, in the form of **`is`** and **`is not`**. **`==`** and **`!=`** are

used to compare values, but is and is not compare references. (For example, whether two variables refer to a single object). A special use is comparison to **`None`** type and **`NotImplemented`** type, like **`if a is not Null`** or **`if __exec__() is NotImplemented`**.

## 10. Miscellaneous Facts

- Python comments begin with **`#`**, and supports single-line comments only. There is no concept of block comments in Python.
- Variable names can contain letters, numbers and underscores. They cannot begin with a number.
- Escape sequences are exactly the same as C.
- Avoid starting the variable name with underscore, this is usually reserved for libraries and language defaults.
- Two string literals put next to each other are automatically concatenated. Example:

```
lang = “Py””thon”
print(lang)          # prints “Python”
```

- Triple-quotes are used to pre-format strings over several lines. Example:

```
print("""\
Usage: thingy [OPTIONS]
     -h                   Display this usage message
     -H hostname          Hostname to connect to
""")
```

- The _ variable holds the result of the most recent operation. Example:

```
width = 20
length = 30
area = width * length
height = 10
print(“Volume is “, _ * height) #prints “Volume is 600”
```

- Strings can be indexed. Negative indices start counting from the right. String index ranges can also be used, like below.

```
word = “Python”              # word[0] is ‘P’
# word[4] is ‘o’             # word[-6] is ‘P’
# word[-1] is ‘n’            # word[:] is ‘Python’
# word[0:2] is ‘Py’          # word[3:5] is ‘hon’
# word[2:] is ‘thon’         # word[:4] is ‘Pytho’
```

- Strings are immutable. For example, if **`str=“Python”`**, you cannot use **`str[0]=‘J’`**.
- Raw strings (where you do not want to escape characters) have an **`r`** prefix to them, and Unicode strings have a **`u`** prefix to them. Example,

```
print(r'C:\some\name')          #prints “C:\some\name”
```

- Similarly, sending byte sequences have a **b** prefix. (ASCII only, 0-255). This is used in rare cases, and cannot be treated as or concatenated with strings. Example:
  ```
  print(b'\x41\x37')        #prints b'A7'
  ```
- There are three useful functions used in association with strings. **`rstrip(`*`[character]`*`)`** removes the specified character from either end of the string, and removes spaces if no argument is specified. ***`string`*`.split(`*`[delimiter]`*`)`** splits a string into a list separated by the specified delimiter, and ***`delimiter`*`.join(`*`list`*`)`** joins all the elements of the list by the specified delimiter string.

## 11. Decision making

Decision making statements are fairly straightforward in Python. It differs from C in the following aspects:

- No parentheses enclose conditions
- Blocks are indented
- else-if is replaced by elif
- Switch-case is not supported in Python

The syntax of the statements are:

### 11.1 if

```
if cond:
        run_this
```

### 11.2 if-else

```
if cond:
        run_this
else:
        run_this
```

### 11.3 if-elif-else

```
if cond:
        run_this
elif cond:
        run_this
else:
        run_this
```

Note that the expression in the **cond** has to evaluate to a Boolean value or be a Boolean value in the first place, by best practices. In theory, only values that evaluate to **False** are integer **0** and **None**.

Also, Python recommends you use the logical operators and avoid nesting if-else conditionals.

## 11.4 try/except blocks

Try / except is a mechanism that causes change of flow on occurrence of an exception or error in the runtime, and should not be used for other purposes to maintain semantical validity and maintainability.

Python provides exception handling using try/except blocks. Syntax is:

```
try:
        try_block
        #throw
except ([exception_type1],[exception_type2],[exception_type3]):
        exception_handler
finally:
        cleanup_action
```

The try block contains the "risky code", where there is a possibility of having an error. For example, you might be trying to open a file, and there is a possibility that the file may not exist. In such a case, you put the code to open the file in the try block, specify **FileExistsError** in the **except** statement, and write code to handle the error – say, tell the user the file was not found. This mechanism ensures that your program "handles" and recovers from an error that happens, and not crash silently in the background.

The **except** block executes only if the specified type of error / exception happens, otherwise, it is skipped. The **finally** block contains code that is always executed, despite whether the error / exception occurred or not. For example, you will need to release the connection to the database whether or not the insert operation fails.

If you are writing custom code, you can create your own exceptions for debugging, by creating exception classes that inherit from the **Exception** class (defined by Python).

On a side note, **raise *exception_type([friendly_message])*** is used to create and throw exceptions, system-defined or user-defined.

## 12. Iteration

Iteration repeats a set of instructions based on a condition. The condition should be a Boolean expression, or follow Boolean conventions as explained in primitives. Care should be taken to avoid infinite loops. Iteration or looping falls into three categories in Python

### 12.1 For loop using range

This is the "traditional" for loop, that runs over a defined numeric range. The expression after **in** should be an iterable type in Python

```
for x in range(r1,r2):
        run_this
```

**r1** and **r2** are integers, like **for i in range(0,10):**

### 12.2 For loop over a iterable

This construct iterates over a set of items in a list or dictionary.

```
for var in collection:
        run_this
```

For example,

```
for word in words:      # words = ["My","name","is","HEISENBERG"]
        capitalize(word)
```

### 12.3 While loop (Also, indefinite loop)

While loop is similar to the one in C, with the following syntax:

```
while cond:
       run_this
       cond_satisfy        #make sure the while loop terminates
```

Note that Python does *NOT* support **do-while** loop.

### 12.4 Break, Continue

The conditions for iteration support statements are simple.

- **break** stops the loop and jumps to the next instruction
- **continue** stops the execution and checks the condition again, without finishing the rest of the loop body

## 13. Functions

Functions provide a mechanism for modularity and reuse, by allowing code to "jump" to a different point and return back to the jump point after performing some computation. Parameters and return values are optional. **def** is used to "define" a function. In addition to those you define, Python defines many built-in functions like **print()**, **min()**, **length()** and some library functions like **math.random()** and **math.sin()**. Python's syntax for defining a function is:

```
def func_name([params]):
      do_something_here
      do_more_here
      return [something]
```

You can specify default values for function parameters, (including **None**), as shown:

```
def add(a=0,b=0):
   return a+b  # returns 0 if add() is called, 6 if add(6) is called
```

Also, you can change the order of the parameters, but you need to specify the names of the arguments along with the values. For example, if a function **def login (username, password, url)** is defined, then, it is legal to call the **login(url='http://some_url/',username='user',password='r-crYp1:')**

There are two types of functions, depending on the return type. The third type is called Lambda function, and is akin to Macros in C.

## 13.1 Void functions

These functions have a **return** at the end. They do not return any data to the caller, only the control is returned.

## 13.2 Fruitful functions

These functions can return data to the caller, in any form. The return value may be a primitive, list, dictionary or any other Python Object. The **return** at the end is followed by the value / variable that needs to be returned.

## 13.3 Lambdas

Lambdas are anonymous functions that do not follow the same syntax as a normal function. In many places (in spirit of functional programming), lambdas can be passed to other functions like any other parameter. Example:

```
f = lambda x: x**2      # print(f(8)) prints 64
foo = [2, 18, 9, 22, 17, 24, 8, 12, 27]
print (filter(lambda x: x%3==0, foo)) # prints [18, 9, 24, 12, 27]
```

## 14. Object Orientation in Python

Like most modern languages, Python supports Object Oriented philosophy. Even though it does not force Java's "Everything is an object" model, internally, everything in Python is an object. This allows maximum flexibility for the programmer.

All the classic OOP constructs like encapsulation, polymorphism, inheritance and abstraction are supported. Let us consider the program below:

```
class Employee:
   """Common base class for all employees"""      # class documentation
   empCount = 0                                    # class variable

   def __init__(self, name, salary):               # constructor
      self.name = name
      self.salary = salary
      Employee.empCount += 1

   def displayCount(self):                          #class methods
     print("Total Employees:%d" %Employee.empCount) # using C-style print

   def displayEmployee(self):
      print("Name:",self.name,",Salary:",self.salary) # Python style print

emp1 = Employee("Zara", 2000)              # object initializations
emp2 = Employee("Joe", 1500)
emp1.displayCount()                        # prints 2
emp2.displayEmployee()                     #prints Name:Joe,Salary:1500
```

- The keyword **`class`** tells Python that we are defining a class. The class name follows, followed by the class block.
- Triple quoted strings inside the class represent 'Class Documentation' that can be accessed by **`classname.__doc__`** variable. This is a good practice to write documentation for each class you define in Python.
- Class members and member functions are places inside the class block. The normal variable naming conventions and rules apply.
- **`self`** is the reference to the object, like the **`this`** keyword in C and Java. Python requires you to explicitly mention this variable in each class function, unless it is a static function (not associated with the class instance). Note that **`self`** is not a keyword, you can use this or any other valid variable name in place of it.
- The **`__init__()`** function is the constructor, and is called each time you instantiate an object of that class. Similarly, there exists the **`__del__()`** function for destructor, though it is not used frequently (Python manages garbage collection internally).

- Variables that begin with double underscores (__) are protected class members. Others are public by default.
- Inheritance works by inheriting from a parent class or a set of parent classes. The syntax is:

```
class SubClassName (ParentClass1[, ParentClass2, ...]):
                “””Optional class documentation string”””
                # class_suite
```

- Method overriding is achieved by having a function with the same name in the child class. The nearest function (child precedence) is called.
- Operator overriding is achieved by overriding the **`__add__()`**, **`__sub__()`**, **`__radd__()`**, **`__lt__()`**, **`__gt__()`**, **`__eq__()`**, **`__iadd__()`**, etc., These are built-in functions that are executed in lieu of operators for classes. The names surrounded by __ are called *magic methods* or *dunder methods*, and **`__init__()`** is pronounced "*dunder init dunder*"

## 15. Imports

We grew out of single-file source codes a long ago. As programs get more complex, in view of maintainability and logical organization, we will need to "break" the code over several files, in order to separate functionalities. In fact, software architecture defines patterns like MVVM and MVC (Model-View-Controller) that suggest separation of concerns. Also, libraries that you import into Python should have a mechanism to be included into the program, like the C **`#include<>`** or Java **`include`**.

Python uses the import statement to reference and use code from other files. Printing an imported module will print its location, like so:

```
import math
print math # prints <module 'math' from '/usr/lib/python2.5/lib-
             dynload/math.so'>
```

Finally, you can import either modules or libraries. To import a library, a plain **`import`** ***`library_name`*** is used, whereas, to import a specific module, **`from`** ***`library_name`*** **`import`** ***`module_name`*** or **`import`** ***`library_name.module_name`*** is used. For the second approach, remember that the library name prefix is mandatory whenever the module is called, because it is not imported to the program namespace.

Python is not pre-processed. If you want to, say, read constants from a file constants.py, you can just place it in the same directory and just say import constants to be able to access variables in that file.

To create your own libraries, you need to create a python package (basically, a directory) with a file called **`__init__.py`**. This file decides which modules are exposed as APIs, while keeping other modules internal. This is done by overriding the **`__all__`** list, including the names of the exposed modules.

## 16. File Operations and Persistence

Persistence is an important property, where you preserve changes outside the program. Without persistence, every time you stop program execution, it starts from its initial state, which is not desirable for most applications.

With this in mind, the most used persistence techniques are writing to flat files, followed by databases. We will get to databases later, and look at writing to files and reading from files. The program needs a "handle" to deal with files. Handle is basically a variable that points to the file in order to perform operations on it.

```
handle = open(file_name,[mode],[buffering])
```

The file name follows the standard relative / absolute convention, and mode is read (by default), or write(**w**). Other modes like binary(**b**), append(**a**), initial(**+**) can be used in groups. The buffering argument is **0** or **1** to disable or enable buffering respectively.

To read from a file, ***handle*.read(*[byte_count]*)** can be used. The optional byte_count argument specifies how many bytes needs to be read from the file, and defaults to all contents of the file.

To write to a file, ***handle*.write(*Content_string*)** can be used. As expected, it writes the specified string to the file. Nothing more to it. Sometimes, you will notice that your contents are not reflected in the actual file. Why? Because it is still in the buffer. To write to the file, you need to **close()** it if you are done, or **flush()** it if you need to perform more operations.

While we are discussing file operations, the **os** library supports file renames, deletions, and working with directories.

```
os.rename(cur_file_name, new_file_name)   # rename a file
os.mkdir(new_dir_name)                    # create new directory
os.chdir(dir_name)                   # change to specified directory
os.getcwd()                          # returns current working directory
os.rmdir('dirname')                  # delete a directory
```

## 17. PIP – PIP Installs Packages

PIP is a command-line package management system used to install and manage software packages in Python. Installing and removing libraries you require is managed by PIP. PIP is installed with Python by default. The syntax for installing/removing a package is:

```
pip install some-package-name
pip uninstall some-package-name
```

Installed PIP libraries can by recreated with corresponding version numbers for use with another computer / cloud system or virtual environment (**venv**) by using

```
pip install -r requirements.txt
```

Internally, PIP downloads something called a Wheel package by auto-detecting your Python version number and dependencies for the current library and installs them.

## 18. PyMySQL library

Another major form of persistence is the use of databases. Databases are better for more complex structuring of data, and can handle constraints and indexing across multiple processes and threads. Also, retrieving data is much easier by the use of querying languages.

ORM (Object-Relation Models) allow direct translation of an "object" in a programming language to a database entry. A popular ORM framework for Python is SQLAlchemy.

We will be looking at MySQL in particular, for this demonstration. Install the access library by opening the command line and using **`pip install pymysql`**. Once that is done, the library can be imported into any Python program by using the **`import pymysql`** statement at the beginning of your program. Note that you should have set up MySQL previously, and know the database name, username and password (this typically constitutes the "connection string") to connect to the database.

To connect to the database, we make a connection using a regular Python variable:

```
con = pymysql.connect('localhost', 'root', 'pavap', 'test')
```

Next, you have to use a 'cursor', to point to this connection and perform query operations on the database. **`cur = con.cursor()`**

Using this cursor, it is easy to execute SQL queries. **`cur.execute(`*`SQL_Query`*`)`**

## 18.1 Reading queries

As soon as you execute a select query (that retrieves results from a database, if any) using **`cur.execute()`**, the results returned are stored in the cursor, and can be accessed via the **`cur.fetchone()`** or **`cur.fetchall()`** functions to return one or all results (which can be further accessed by a **`for..in`** loop).

## 18.2 Create / Update / Delete queries

A simple **`cur.execute()`** with insert / update / delete query will achieve the task of modifying the database, provided the credentialed user has such a permission.

## 18.3 Best practices – an example

```
import pymysql

# Connect to the database using connection string
connection = pymysql.connect(host='server_name_or_url',
                             user='username',
                             password='passwd',
                             db='database_name')

# Surround with try / except block, database operations are risky!
try:
    with connection.cursor() as cursor:
        sql="INSERT INTO `users`(`email`, `password`) VALUES (%s, %s)"
        cursor.execute(sql, ('webmaster@python.org', 'very-secret'))

     # connection is not autocommit by default, perform commit to save
     connection.commit()

    with connection.cursor() as cursor:
        # Read a single record
        sql = "SELECT `id`, `password` FROM `users` WHERE `email`=%s"
        cursor.execute(sql, ('webmaster@python.org',))
        result = cursor.fetchone()
        print(result)
except Error: # handle error here
finally:
    connection.close()

# Result is: {'password': 'very-secret', 'id': 1}
```

## 19. BeautifulSoup library

Sometimes, we have to face the fact that we do not have access to the database. For instance, you cannot expect Amazon, eBay and Flipkart to give you access to their catalog and price database so you can write a script to compare prices before you buy, right? Similarly, your university might not give you a database access to find out your classmates' marks and compare them. In such cases, where information is available on a webpage, you can "scrape" the webpage by writing a "spider" and "crawling" through them.

First off, in order to scrape a webpage, it is advisable to have a basic knowledge of HTML tag structure and supported attributes. You can look up w3schools to know more about HTML tags.

BeautifulSoup allows you to "read" a webpage by its tags, IDs, etc., and filter out the information for further processing. Even though you can write a web parser by reading page contents, BeautifulSoup handles malformed tags and generic attributes well.

**`from bs4 import BeautifulSoup`** will import the BeautifulSoup library. It makes use of a "**`soup`** object" that lets you read components from the page. **`soup = BeautifulSoup(html_doc, 'html.parser')`**. **`html_doc`** is the file handle / URL handle to the webpage, and the second argument is optional, indicating which parser to use. **`lxml`**, **`lxml-xml`**, **`html5lib`** can be used.

Tag contents can be read easily by using the dot operator with **`soup`** object. For example, **`soup.title`** prints the title of the page. You can also navigate as **`soup.article.em`**, where it searches for occurrences of tag **`em`** inside **`article`** elements only. Note that this finds only the first occurrence of the tag. For example, to find the first link in a page, use **`soup.a`**. Instead, if you like to find all the links on a page, use **`soup.find_all('a')`**, it returns a list with all the links. To find a tag with a particular ID (say, price), you can use **`soup.find(id="price")`**

If you want to print a particular attribute of the tag, you can use the .get method. For example, printing all the links in a page, you can use:

```
for link in soup.find_all('a'):
    print(link.get('href'))
```

To print the contents, **`soup.get_text()`** is useful. If you want to pretty-print, **`soup.prettify()`** will properly indent the lines. The following table lists other often-used BeautifulSoup functions, but for a full list, refer the documentation.

| tag…. | **Returns** |
|---|---|
| `.string` | the bit of string within the tag |
| `.attrs` | dictionary of attributes and values |
| `.name` | name of the tag |
| `.contents` | list with all content tags, direct. |
| `.children` | same as **`.contents`**, but iterable |
| `.strings` | all the contents (no tags) of specified tag. Use **`.stripped_strings`** to remove extra whitespaces |
| `.descendants` | list of children, recursively |
| `.parent` | Returns immediate parent of the tag. **`.parents`** returns a list of parents recursively. |
| `.next-sibling, .previous-sibling` | Navigate between page elements on the same level of the parse tree. ….**`.siblings`** returns iterables. |
| `.has_attr()` | **`True`** if tag has the mentioned attribute, **`False`** otherwise |
| `.select(CSS_selector)` | list of tags that contain the specified selector |

| `.find_all(…)` | **Returns** |
|---|---|
| `("p","title")` | all **p** and **`title`** tags |
| `("a", attrs={"class": "danger"})` | all **a** tags which have the class "**`danger`**" |
| `(id="description")` | this is a filter, searches attributes where the **`id`** attribute is set to "**`description`**" |
| `(p, limit=2)` | **p** tags, maximum of (first) 2 in the list |
| `(p, recursive=True)` | Recursively lists **p** tags (**p** tags within **p** tags are also searched) |
| `("p", class_="body str")` | **p** tags, where CSS class is "**`body str`**" only, and not "**`str body`**" |

If you're having trouble understanding what Beautiful Soup does to a document, pass the document into the **`diagnose()`** function. Beautiful Soup will print out a report showing you how different parsers handle the document, and tell you if you're missing a parser that Beautiful Soup could be using:

```
from bs4.diagnose import diagnose
data = open("bad.html").read()
diagnose(data)
```

## 20. Being Pythonic

Over time, as the Python language evolved and the community grew, a lot of ideas arose about how to use Python the right way. The Python language actively encourages a large number of idioms to accomplish a number of tasks ("the one way to do it"). In turn, new idioms that evolved in the Python community has have in turn influenced the evolution of the language to support them better. Consider the following example:

```
i=0
while i<list.length:
   function(mylist[i])
   i+=1
```

```
for i in
range(list_length):
   function(mylist[i])
```

```
for element in list:
   function(element)
```

All three snippets perform the same function, but the "recommended" way to do it is the one in the end. Similarly, the example shows how to exploit tuples to achieve the task, instead of using a verbose function.

```
def foo(a, b):
    a[0] = 3
    b[0] = 5.5
alpha = [0]
beta = [0]
foo(alpha, beta)
alpha = alpha[0]
beta = beta[0]
```

```
def foo():
    return 3,5.5

alpha,beta = foo()

# Pythonic!
```

Our point is, code that is not "Pythonic" tends to look odd or cumbersome to an experienced Python programmer. It may also be harder to understand, as instead of using a common, recognizable, brief idiom, another, longer, sequence of code is used to accomplish the desired effect. Since the language tends to support the right idioms, non-idiomatic code frequently also executes more slowly.

Remember that Python was built around the core of understandability and simplicity – and in order to achieve that, there have been more efficient data structures and functions added over the years that let you concentrate on the task with the least amount of coding effort, at the same time, not sacrificing understandability.

These idioms extend beyond programming constructs, and you should keep them in mind when you are, say, writing a library. It is your duty to make the code as easy and natural as possible for a Python programmer to pick up and perform a task. Also, as you become more experienced in Python programming, you will realize the importance of the possibilities that you can exploit by utilizing these idioms, like passing methods to functions. Dedicate some time to tell your story the pythonic way!