Project ⋆ Python

Nguyen Quan Ba Hong
Students at Faculty of Math and Computer Science,
Ho Chi Minh University of Science, Vietnam
email. nguyenquanbahong@gmail.com
blog. http://hongnguyenquanba.wordpress.com [1]

October 6, 2016

Contents

1 Basics **1**

1.1 Print Commands 1
1.2 Indentation 1
1.3 Variables and Types 1
1.3.1 Numbers 2
1.3.2 Strings 2
1.4 Lists 2
1.5 Basic Operators 3
1.5.1 Arithmetic Operators 3
1.5.2 Using Operators with Strings 3
1.5.3 Using Operators with Lists 3
1.6 String Formatting 3
1.7 Basic String Operators 4
1.8 Conditions 5
1.8.1 Boolean Operators 5
1.8.2 The `in` Operator 5
1.8.3 The `is` Operator 6
1.8.4 The `not` Operator 6
1.9 Loops 6
1.9.1 `for` Loop 7
1.9.2 `while` Loops 7
1.9.3 `break` and `continue` Statements 7
1.9.4 `else` for Loops 8
1.10 Functions 8
1.10.1 Write Functions in Python 8
1.10.2 Call Functions in Python 9
1.11 Classes and Objects 9
1.11.1 Accessing Object Variables 9
1.11.2 Accessing Object Functions 9
1.12 Dictionaries 10
1.12.1 Iterating Over Dictionaries 10
1.12.2 Removing a value 10
1.13 Modules and Packages 10
1.13.1 Exploring Built-in Modules 11
1.13.2 Writing Modules 11
1.13.3 Writing Packages 12

2 Advanced Tutorials **13**
2.1 Generators 13
2.2 List Comprehensions 13
2.3 Multiple Function Arguments 14
2.4 Regular Expressions 15
2.5 Exception Handling 15
2.6 Sets 15
2.7 Serialization 16
2.8 Partial Functions 17
2.9 Code Introspection 17
2.10 Closures 18
2.11 Decorators 19

Chapter 1

Basics

1.1 Print Commands

Python is a very simple language, and has a very straightforward syntax. It encourages programmers to program without boilerplate (prepared) code. The simplest directive in Python is the "print" directive. It simply prints out a line (and also includes a newline, unlike in C).

There are two major Python versions, Python 2 and Python 3. Python 2 and 3 are quite different. Python 2 is more widely used and supported. However, Python 3 is more semantically correct, and supports newer features.

For example, one difference between Python 2 and 3 is the `print` statement. In Python 2, the `print` statement is not a function, and therefore it is invoked without parentheses. However, in Python 3, it is a function, and must be invoked with parentheses.

To print a string, just write

```
print "This line will be printed"
```

in Python 2 and

```
print('This line will be printed')
```

1.2 Indentation

Python uses indentation for blocks, instead of curly braces. Both tabs and spaces are supported, bu the standard indentation requires standard Python code to use four spaces.

1.3 Variables and Types

Python is completed object oriented, and not "statically typed". You do not need to declare variables before using them, or declare their type. Every variable in Python is an object.

1.3.1 Numbers

Python supports two types of numbers - integers and floating point numbers. It also supports complex numbers.

To define an integer, use the following syntax

```
myint = 7
```

To define a floating point number, you may use one of the following notations

```
myfloat = 7.0
myfloat = float(7)
```

1.3.2 Strings

Strings are defined either with a single quote or a double quotes.

```
mystring = 'hello'
mystring = "hello"
```

The difference between the two is that using double quotes makes it easy to include apostrophes (whereas these would terminate the string if using single quotes)

```
mystring = "Don't worry about apostrophes"
```

There are additional variations on defining strings that make it easier to include things such as carriage returns, backslashes and Unicode characters. See in [2].

Assignments can be done on more than one variable "simultaneously" on the same line like this

```
a,b = 3,4
```

Mixing operators between numbers and strings is not supported.

1.4 Lists

Lists are very similar to arrays. They can contain any type of variable, and they can contains as many variables as you wish. Lists can also be iterated over in a very simple manner. Here is an example of how to build a list.

```
mylist = []
mylist.append(1)
mylist.append(2)
mylist.append(3)
print(mylist[0]) # prints 1
print(mylist[1]) # prints 2
print(mylist[2]) # prints 3

# prints out 1,2,3
for x in mylist
    print x
```

Accessing an index which does not exist generates an exception (am error).

```
mylist = [1,2,3]
print(mylist[10])
```

1.5 Basic Operators

This section explains how to use basic operators in Python.

1.5.1 Arithmetic Operators

Just as any other programming languages, the addition, subtraction, multiplication and division operators can be used with numbers.

Another operator available is the modulo % operator, which returns the integer remainder of the division. dividend% divisor = remainder.

```
remainder = 10 % 3
```

Using two multiplication symbols makes a power relationship.

```
squared = 7 ** 2
cubed = 2 ** 3
```

1.5.2 Using Operators with Strings

Python supports concatenating strings using the addition operator.

```
a b = "a" + " " + "b"
```

Python also supports multiplying strings to form a string with a repeating sequence.

```
lotsofa = "a" * 10
```

1.5.3 Using Operators with Lists

Lists can be joined with the addition operators.

```
even_digits = [0,2,4,6,8]
odd_digits = [1,3,5,7,9]
all_digits = even_digits = odd_digits
```

Just as in strings, Python supports forming new lists with a repeating sequence using the multiplication operator.

```
print [1,2,3] * 3
```

1.6 String Formatting

Python uses C-style string formatting to create new, formatted strings. The % operator is used to format a set of variables enclosed in a “tupled” (a fixed size list), together with a format string, which contains normal text together with “argument specifiers”, special symbols like “%s” and “%d”.

```
name = "Hong"
print "Hello, %s" % name
```

To use two or more argument specifiers, use a tuple (parentheses).

```
name = "Hong"
age = 20
print "%s is %d years old." % (name, age)
```

Any object which is not a string can be formatted using the %s operator as well. The string which returns from the "repr" method of that object is formatted as the string. For example

```
mylist = [1,2,3]
print "A list: %s" % mylist
```

Here are some basic argument specifiers you should know.

1. `%s` String (or any object with a string representation, like numbers).
2. `%d` Integers.
3. `%f` Floating point numbers.
4. `%.<number of digits>f` Floating point numbers with a fixed amount of digits to the right of the dot.
5. `%x/%X` Integers in hex representation (lowercase/uppercase).

1.7 Basic String Operators

Strings are bits of text. They can be defined as anything between quotes.

```
astring = "Python"
astring2 = 'Python'
```

As you can see, the first thing you learned was printing a simple sentence. This sentence was stored by Python as a string. However, instead of immediately printing string out, we will explore the various things you can do to them. You can also use the single quotes to assing a string. However, you will face problems if the value to be assigned itself contains single quotes. For example to assign the string in these bracket, you need to use double quotes only like this

```
print "single quotes are ' ' "
print len(astring)
print astring.index("o") # only recognizes the first
print astring.count("P")
print astring[2:5]
```

This prints a slice of the string, starting at index 2 and ending at index 4. Most programming languages do this. It makes doing math inside those brackets easier.

If you just have one number in the brackets, it will give you the single character at that index. If you leave out the first number but keep the colon, it will give you a slice from the start to the number you left in. If you leave out the second number, it will give you a slice from the first number to end.

You can even put negative numbers inside the brackets. They are an easy way of starting at the end of the string instead of the beginning. This way, `-3` means "3rd character from the end".

```
print astring[start:stop:step]
```

This prints the characters of string from `start` to `stop` skipping `step` characters.

There is no function like `strrev` in C to reverse a string. But with the above mentioned type of slice syntax you can easily reverse a string like this

```
print astring[::-1]
```

This

```
print astring.upper()
print astring.lower()
```

make a new string with all letters converted to uppercase and lowercase, respectively.

```
print astring.startswith("Hello")
print astring.endswith("Bye")
```

This is used to determine whether the string starts with something or ends with something, respectively. The first one will print `True`, as the string starts with `"Hello"`. The second one will print `False`, as the string certainly does not end with `"Bye"`.

```
afewwords = astring.split(" ")
```

This splits the string into a bunch of strings grouped together in a list.

1.8 Conditions

Python uses boolean variables to evaluate conditions. The boolean values `True` and `False` are returned when an expression is compared or evaluated.

Notice that variable assignment is done using a single equals operators `=`, whereas comparison between two variables is done using the double equals operator `==`. The "not equals" operator is marked as `!=`.

1.8.1 Boolean Operators

The `and` and `or` boolean operators allow building complex boolean expressions.

```
if x == 1 and c == "A":
    print('x = 1, c = A')
if x == 1 or c == "A":
    print('x = 1 or c = A')
```

1.8.2 The `in` Operator

The `in` operator could be used to check if a specified object exists within an iterable object container, such as a list

```
if c in ["a","A"]:
    print('x is a or A')
```

Python uses indentation to define code blocks, instead of brackets. The standard Python indentation is 4 spaces, although tabs and any other space size will work, as long as it is consistent. Notice that code blocks do not need any termination.

Here is an example for using Python's `if` statement using code blocks

```
if <statement is true>:
    <do something>
    ...
elif <another statement is true>: # else if
    <do something else>
    ...
else:
    <do another thing>
    ...
```

A statement is evaluated as true if one the the following is correct.

1. The `True` boolean variable is given, or calculated using an expression, such as an arithmetic comparision.
2. An object which is not considered "empty" is passed.

Here are some examples for objects which are considered as empty.

1. An empty string `""`.
2. An empty list `[]`.
3. The number zero `0`.
4. The false boolean variable `False`.

1.8.3 The `is` Operator

Unlike the double equals operator `==`, the `is` operator does no match the values of variables, but the instances themselves. For example

```
x = [1,2,3]
y = [1,2,3]
print x == y # Prints out True
print x is y # Prints out False
```

1.8.4 The `not` Operator

Using `not` before a boolean expression inverts it.

```
print not False # Prints out True
print (not False) == (False) # Print out False
```

1.9 Loops

There are two types of loops in Python, `for` and `while`.

1.9.1 `for` Loop

For loops iterate over a given sequence. For example

```
primes = [2,3,5,7]
for prime in primes:
    print prime
```

For loops can iterate over a sequence of numbers using the `range` and `xrange` functions. The difference between `range` and `xrange` is that the `range` function returns a new list with numbers of that specified range, whereas `xrange` returns an iterator, which is more efficient. Python 3 uses the `range` function, which acts like `xrange`. Note that the `xrange` function is zero based.

```
# Prints out the numbers 0,1,2,3,4
for x in xrange(5): # or range(5)
    print x
# Prints out 3,4,5
for x in xrange(3,6): # or range(3,6)
    print x
# Prints out 3,5,7
for x in xrange(3,8,2): # or range(3,8,2)
    print x
```

1.9.2 `while` Loops

`While` loops repeat as long as a certain boolean condition is met. For example

```
# Prints out 0,1,2,3,4
count = 0
while count < 5:
    print count
    count += 1
```

1.9.3 `break` and `continue` Statements

`break` is used to exit for a loop or a `while` loop, whereas `continue` is used to skip the current block, and return to the `for` or `while` statement. A few examples

```
# Prints out 0,1,2,3,4
count = 0
while True:
    print count
    count += 1
    if count >= 5:
        break
# Prints out only odd numbers 1,3,5,7,9
for x in xrange(10):
    if x % 2 == 0:
        continue
    print x
```

1.9.4 `else` for Loops

Unlike languages like C, C++, etc., we can use `else` for loops. When the loop condition of `for` or `while` statement fails then code part in `else` is executed. If `break` statement is executed inside for loop then the `else` part is skipped. Not that `else` part is executed if there is a `continue` statement. For example

```
# Prints out 0,1,2,3,4 and then it prints "count value reached 5"
count = 0
while(count < 5):
    print count
    count += 1
else:
    print "count value reached %d" %(count)
# Prints out 1,2,3,4
for i in xrange(1,10):
    if(i % 5 == 0):
        break
    print i
else:
    print "this is not printed because for loop is terminated."
```

1.10 Functions

Functions are a convenient way to divide your code into useful blocks, allowing us to order our code, make it more readable, reuse it and save some time. Also functions are a key way to define interfaces so programmers can share their code.

1.10.1 Write Functions in Python

Python makes use of blocks.

A block is a are of code of written in the format of

```
block_head:
    1st block line
    2nd block line
    ...
```

Where a bloc line is more Python code (even another block), and the block head is of the following format

```
block_keyword
key_name(argument1,argument2,...)
```

Block keywords you already know are `if`, `for` and `while`.

Functions in Python are defined using the block keyword `def`, followed with the function's name as the block's name. For example

```
def my_function():
    print "Hi"
```

Functions may also receive arguments (variables passed from the caller to the function).

Functions may return a value to the caller, using the keyword `return`. For example

```
def sum_two_numbers(a,b):
    return a + b
```

1.10.2 Call Functions in Python

Simply write the function's name followed by (), placing any required arguments within the brackets.

```
myfunction()
x = sum_two_numbers(1,2)
```

1.11 Classes and Objects

Objects are an encapsulation of variables and functions into a single entity. Objects gets their variables and functions from classes. Classes are essentially a template to create your objects.

A very basic class would look something like this

```
class MyClass
    variable = ...
    def function(f):
        ...
```

We will explain why you have to include that "self" as a parameter a little bit later. First, to assign the above class (template) to an object you would do the following.

```
myobjectx = MyClass()
```

Now the variable `myobjectx` holds an object of the class `MyClass` that contains the variable and the function defined within the class called `MyClass`.

1.11.1 Accessing Object Variables

To access the variable inside of the newly created object `myobjectx` you would do the following.

```
myobjectx.variable
```

You can create multiple different objects that are of the same class (have the same variables and functions defined). However, each object contains independent copies of the variables defined in the class.

1.11.2 Accessing Object Functions

To access a function inside of an object you use notation similar to accessing a variable

```
myobjectx.function()
```

1.12 Dictionaries

A dictionary is a data type similar to arrays, but works with keys and values instead of indexes. Each value stored in a dictionary can be accessed using a key, which is any type of object (a string, a number, a list, etc.) instead of using its index to address it.

For example, a database of phone numbers could be stored using a dictionary like this

```
phonebook = {}
phonebook["John"] = 938477566
phonebook["Jack"] = 938377264
phonebook["Jill"] = 947662781
```

Alternatively, a dictionary can be initialized with the same values in the following notation

```
phonebook = {
    "John" : 938477566,
    "Jack" : 938377264,
    "Jill" : 947662781
}
```

1.12.1 Iterating Over Dictionaries

Dictionaries can be iterated over, just like a list. However, a dictionary, unlike a list, does not keep the order of the values stored in it. To iterate over key value pairs, use the following syntax

```
for name, number in phonebook.iteritems():
    print "Phone number of %s is %d" %(name,number)
```

1.12.2 Removing a value

To remove a specified index, use either one of the following notations

```
del phonebook["John"]
```

or

```
phonebook.pop("John")
```

1.13 Modules and Packages

Modules in Python are simply Python files with the `.py` extension, which implement a set of functions. Modules are imported from other modules using the `import` command.

To import a module, we use the `import` command. Check out the full list of built-in modules in the Python standard library in [3].

The first time a module is loaded into a running Python script, it is initialized by executing the code in the module once. If another module in your code imports the same module again, it will not be loaded twice but once only - so

local variables inside the module act as a "singleton" - they are initialized only once.

If we want to import the module `urllib`, which enables us to create and read data from URLs, we simply `import` the module

```
# import the library
import urllib
# use it
urllib.urlopen(...)
```

1.13.1 Exploring Built-in Modules

Two very important functions come in handy when exploring modules in Python - the `dir` and `help` functions.

We can look for which functions are implemented in each module by using the `dir` function.

```
>>> import urllib
>>> dir(urllib)
['ContentTooShortError', 'FancyURLopener', 'MAXFTPCACHE',
'URLopener', '__all__', '__builtins__', '__doc__', '__file__',
'__name__', '__package__', '__version__', '_ftperrors',
'_get_proxies', '_get_proxy_settings', '_have_ssl', '_hexdig',
'_hextochr', '_hostprog', '_is_unicode', '_localhost', '_noheaders',
'_nportprog', '_passwdprog', '_portprog', '_queryprog', '_safe_map',
'_safe_quoters', '_tagprog', '_thishost', '_typeprog', '_urlopener',
'_userprog', '_valueprog', 'addbase', 'addclosehook', 'addinfo',
'addinfourl', 'always_safe', 'basejoin', 'c', 'ftpcache',
'ftperrors', 'ftpwrapper', 'getproxies', 'getproxies_environment',
'getproxies_macosx_sysconf', 'i', 'localhost', 'main', 'noheaders',
'os', 'pathname2url', 'proxy_bypass', 'proxy_bypass_environment',
'proxy_bypass_macosx_sysconf', 'quote', 'quote_plus', 'reporthook',
'socket', 'splitattr', 'splithost', 'splitnport', 'splitpasswd',
'splitport', 'splitquery', 'splittag', 'splittype', 'splituser',
'splitvalue', 'ssl', 'string', 'sys', 'test', 'test1', 'thishost',
'time', 'toBytes', 'unquote', 'unquote_plus', 'unwrap',
'url2pathname', 'urlcleanup', 'urlencode', 'urlopen', 'urlretrieve']
```

When we find the function in the module we want to use, we can read about it more using the `help` function, inside the Python interpreter.

```
help(urllib.urlopen)
```

1.13.2 Writing Modules

Writing Python modules is very simple. To create a module of your own, simply create a new `.py` file with the module name, and then import it using the Python file name (without the `.py` extension) using the `import` command.

1.13.3 Writing Packages

Packages are namespaces which contain multiple packages and modules themselves. They are simply directories, but with a twist.

Each package in Python is a directory which must contain a special file called `_init_.py`. This file can be empty, and it indicates that the directory it contains is a Python package, so it can be imported the same way a module can be imported.

If we create a directory called `foo`, which marks the package name, we can then create a module inside that package called `bar`. We also must not forget to add the `_init_.py` file inside the `foo` directory.

To use the module `bar`, we can import it in two ways.

```
import foo.bar
```

or

```
from foo import bar
```

In the first method, we must use the `foo` prefix whenever we access the module `bar`. In the second method, we don't, because we import the module to our module's namespace.

The `_init_.py` file can also decide which modules the packages exports as the API, while keeping other modules internal, by overriding the `_all_` variable, like so

```
__init__.py:

__all__ = ["bar"]
```

Chapter 2

Advanced Tutorials

2.1 Generators

Generators are very easy to implement, but a bit difficult to understand.

Generators are used to create interators, but with a different approach. Generators are simple functions which return an iterable set of items, one at a time, in a special way.

When an iteration over a set of item starts using the statement, the generator is run. Once the generator's function code reaches a "yield" statement, the generator yields its execution back to the `for` loop, returning a new value from the set. The generator function can generate as many values (possibly infinite) as it wants, yielding each one in its turn.

Here is a simple example of generator function which returns 7 random integers.

```
import random
def lottery():
    # returns 6 numbers between 1 and 40
    for i in xrange(6):
        yield random.randint(1,40)
    # returns a 7th number between 1 and 15
    yield random.randint(1,15)
for random_number in lottery()
    print "And the next number is... %d" % random_number
```

This function decides how to generate the random numbers on its own, and executes the yield statements one at a time, pausing in between to yield execution back to the main for loop.

2.2 List Comprehensions

List Comprehensions is a very powerful tool, which creates a new list based on another list, in a single, readable line.

For example, let's say we need to create a list of integers which specify the length of each word in a certain sentence, but only if the word is not the word `the`.

```
sentence = "the quick brown fox jumps over the lazy dog"
words = sentence.split()
word_lengths = []
for word in words
    if word != "the":
        word_lengths.append(len(word))
```

Using a list comprehension, we could simplify this process to this notation.

```
sentence = "the quick brown fox jumps over the lazy dog"
words = sentence.split()
word_lengths = [len(word) for word in words if word != "the"]
```

2.3 Multiple Function Arguments

Every function in Python receives a predefined number of arguments, if declared normally, like this

```
def myfunction(first,second,third):
    # do something with the 3 variables
    ...
```

It is possible to declare functions which receive a variable number of arguments, using the following syntax.

```
def foo(first,second,third,*therest):
    print "First: %s" % first
    print "Second: %s" % second
    print "Third: %s" % third
    print "And all the rest... %s" % list(therest)
```

The `therest` variable is a list of variables, which receives all arguments which were given to the `foo` function after the first 3 arguments.

It is also possible to send function arguments by keyword, so that the order of the argument does not matter, using the following syntax.

```
def bar(first,second,third,**options):
    if options.get("action") == "sum":
        print "The sum is: %d" % (first + second + third)
    if options.get("number") == "first":
        return first
result = bar(1,2,3,action = "sum",number = "first")
print "Result: %d" % result
```

The `bar` function receives 3 arguments. If an additional "action" argument is received, and it instructs on summing up the numbers, then the sum is printed out. Alternatively, the function also knows it must return the first argument, if a `return` argument is received which instructs it.

2.4 Regular Expressions

Regular Expressions (sometimes shortened to `regexp, regex` or `re` are a tool for matching patterns in text. In Python, we have the `re` module. The applications for regular expressions are wide-spread, but they are fairly complex, so when contemplating using a `regex` for a certain task, think about alternatives, and come to `regexes` as a last resort.

An example `regex` is

```
r"^(From|To|Cc).*?python-list@python.org"
```

Now for an explanation: the caret `^` matches text at the beginning of a line. The following group, the part with `(From|To|Cc)` means that the line has to start with one of the words that are separated by the pipe `|`. That is called the `OR` operator, and the `regex` will match if the line starts with any of the words in the group. The `.*?` means to un-greedily match any number of characters, except the newline `\n` character. The un-greedy part means to match as few repetitions as possible. The `.` character means any non-newline character, the `*` means to repeat 0 or more times, and the `?` character makes it un-greedy.

So, the following lines would be matched by that `regex`

```
From: python-list@python.org
To: !asp]<,. python-list@python.org
```

A complete reference for the `re` syntax is available at [4].

2.5 Exception Handling

When programming, errors happen. It's just a fact of life. Perhaps the user gave bad input. maybe a network resource was unavailable. Maybe the program ran out of memory. Or the programmer may have even made a mistake.

Python's solution to errors are exceptions. But sometimes you don't want exceptions to completely stop the program. You might want to do something special when an exception is raised. This is done in a `try/except` block.

For more details on handling exceptions, look no further than [5].

2.6 Sets

Sets are lists with no duplicate entries. Let's say you want to collect a list of words used in a paragraph.

```
print set("my name is Eric and Eric is my name".split())
```

This will print out a list containing `my, name, is, Eric` and finally `and`. Since the rest of the sentence uses words which are already in the set, they are no inserted twice.

Sets are a powerful tool in Python since they have the ability to calculate differences and intersections between other sets. For example, say you have a list of participants in events A and B.

```
a = set(["Jake", "John", "Eric"])
b = set(["John", "Jill"])
```

To find out which members attended both events, you may use the intersection method.

```
>>> a.intersection(b)
set(['John'])
>>> b.intersection(a)
set(['John'])
```

To find out which members attended only one of the events, use the symmetric difference method

```
>>> a.symmetric_difference(b)
set(['Jill', 'Jake', 'Eric'])
>>> b.symmetric_difference(a)
set(['Jill', 'Jake', 'Eric'])
```

To find out which members attended only one event and not the other, use the difference method.

```
>>> a.difference(b)
set(['Jake', 'Eric'])
>>> b.difference(a)
set(['Jill'])
```

To receive a list of all participants, use the union method.

```
>>> a.union(b)
set(['Jill', 'Jake', 'John', 'Eric'])
```

2.7 Serialization

Python provides built-in JSON libraries to encode and decode JSON.

In Python 2.5, the `simplejson` module is used, whereas in Python 2.7, the `json` module is used.

In order to use the `json` module, it must first be imported.

```
import json
```

There are two basic formats for JSON data. Either in a string or the object datastructure. The object datastructure, in Python, consists of lists and dictionaries nested inside each other. The object datastructure allows one to use Python methods (for lists and dictionaries) to add, list, search and remove elements from the datastructure. The String format is mainly used to pass the data into another program or load into a datastructure.

To load JSON back to a data structure, use the `loads` method. This method takes a string and turns it back into the `json` object datastructure.

```
print json.loads(json_string)
```

To encode a data structure to JSON, use the `dump` method. This method takes an object and returns a String.

```
json_string = json.dumps([1,2,3,"a","b","c"])
```

Python supports a Python proprietary data serialization method called pickle (and a faster alternative called cPickle).

You can use it exactly the same way.

```
import cPickle
pickled_string = cPickle.dumps([1,2,3,"a","b","c"])
print cPickle.loads(pickled_string)
```

2.8 Partial Functions

You can create partial functions in Python by using the partial function from the `functools` library.

Partial functions allow one to derive a function with `x` parameters to a function with fewer parameters and fixed values set for the more limited function.

Imported required

```
from functools import partial
```

For example

```
from functools import partial
def multiply(x,y):
    return x*y
# create a new function that multiplies by 2
dbl = partial(multiply,2)
print dbl(4)
```

Note. The default values will start replacing variables from the left. The `2` will replace `x`. `y` will equal 4 when `dbl(4)` is called.

2.9 Code Introspection

Code introspection is the ability to examine classes, functions and keywords to know what they are, what they do and what they know.

Python provides several functions and utilities for code introspection.

```
help()
dir()
hasattr()
id()
type()
repr()
callable()
issubclass()
isinstance()
_doc_
_name_
```

2.10 Closures

A Closure is a function object that remembers values in enclosing scopes even if they are not present in memory.

Firstly, a *Nested Function* is a function defined inside another function. It is very important to note that the nested functions can access the variables of the enclosing scope. However, at least in Python, they are only readonly. However, one can use the "nonlocal" keyword explicitly with these variables in order to modify them.

For example

```
def transmit_to_space(message):
    "This is the enclosing function"
    def data_transmitter():
        "The nested function"
        print(message)
    data_transmitter()
```

This works well as the `data_transmitter` function can access the `message`. To demonstrate the use of the "nonlocal" keyword, consider this

```
def print_msg(number)
    def printer():
        "Here we are using the nonlocal keyword"
        nonlocal number
        number = 3
        print(number)
    print()
    print(number)
print_msg(9)
```

Now, how about we return the function object rather than calling the nested function within. Remember that functions are even object.

```
def transmit_to_space(message):
    "This is the enclosing function"
    def data_transmitter():
        "The nested function"
        print(message)
    return data_transmitter
```

And we call the function as follows

```
>>> fun2 = transmit_to_space("Burn the Sun!")
>>> fun2()
Burn the Sun!
```

Even though the execution of the `transmit_to_space()` was completed, the message was rather preserved. This technique by which the data is attached to some code even after end of those other original functions is called as closures in Python.

Advantage 2.1. Closures can avoid use of global variables and provides some

form of data hiding, e.g., when there are few methods in a class, use closures instead.

Also, Decorators in Python make extensive use of closures.

2.11 Decorators

Decorators allow you to make simple modifications to callable objects like functions, methods and classes. We shall deal with functions for this tutorial. The syntax

```
@decorator
def functions(arg):
    return "Return"
```

is equivalent to

```
def function(arg):
return "Return"
function = decorator(function) # this passes the function to the
# decorator, and reassigns it to the functions
```

As you may have seen, a decorator is just another function which takes a functions and returns one. For example you could do this

```
def repeater(old_function):
    def new_function(*args,**kwds):
ments for how *args and **kwds works
        old_function(*args, **kwds) # Run the old function
        old_function(*args, **kwds) # Do it twice
    return new_function # Have to return the new_function
    # or it wouldn't reassign it to the value
```

This would make a function repeat twice.

```
>>> @repeater
def Multiply(num1, num2):
    print num1*num2

>>> Multiply(2, 3)
6
6
```

You can also make it change the output

```
def Double_Out(old_function):
    def new_function(*args, **kwds):
        # modify the return value
        return 2*old_function(*args, **kwds)
    return new_function
```

change input

```
def Double_In(old_function):
    # only words if the old function has one argument
    def new_function(arg):
        return old_function(arg*2) # modify the argument passed
    return new_function
```

and do checking

```
def Check(old_function):
    def new_function(arg):
        # This causes an error
        # which is better than it doing the wrong thing
        if arg < 0: raise ValueError, "Negative Argument"
        old_function(arg)
    return new_function
```

Let's say you want to multiply the output by a variable amount. You could do

```
def Multiply(multiplier):
    def Multiply_Generator(old_function):
        def new_function(*args, **kwds):
            return multiplier*old_function(*args, **kwds)
        return new_function
    return Multiply_Generator # it returns the new generator
```

Now, you could do

```
@Multiply(3) # Multiply is not a generator, but Multiply(3) is
def Num(num):
    return num
```

You can do anything you want with the old function, even completely ignore it. Advanced decorators can also manipulate the doc string and argument number. For some snazzy decorators, see in [6].

The End

Bibliography

[1] http://www.learnpython.org/

[2] https://docs.python.org/3.6/contents.html

[3] https://docs.python.org/2/library/

[4] https://docs.python.org/3/library/re.html#
regular-expression-syntax"REsyntax

[5] https://docs.python.org/3/tutorial/errors.html#
handling-exceptions

[6] https://wiki.python.org/moin/PythonDecoratorLibrary