Python Functions

A function is a block of code which only runs when it is called. You can pass data, known as parameters, into a function. A function can return data as a result.

```
def my_function():
  print("Hello from a function")
```

Calling a Function

```
def my_function():
  print("Hello from a function")

my_function()
```

Parameters

Information can be passed to functions as parameter.

Parameters are specified after the function name, inside the parentheses. You can add as many parameters as you want, just separate them with a comma.

The following example has a function with one parameter (fname). When the function is called, we pass along a first name, which is used inside the function to print the full name:

```
def my_function(fname):
  print(fname + " Refsnes")

my_function("Emil")
my_function("Tobias")
my_function("Linus")
```

Default Parameter Value

The following example shows how to use a default parameter value. If we call the function without parameter, it uses the default value:

```
def my_function(country = "Norway"):
  print("I am from " + country)

my_function("Sweden")
my_function("India")

my_function()
```

Passing a List as a Parameter

You can send any data types of parameter to a function (string, number, list, dictionary etc.), and it will be treated as the same data type inside the function.

E.g. if you send a List as a parameter, it will still be a List when it reaches the function:

```
def my_function(food):
  for x in food:
    print(x)

fruits = ["apple", "banana", "cherry"]

my_function(fruits)
```

Return Values

To let a function return a value, use the `return` statement:

```
def my_function(x):
  return 5 * x

print(my_function(3))
print(my_function(5))
print(my_function(9))
```

Recursion

Python also accepts function recursion, which means a defined function can call itself.

Recursion is a common mathematical and programming concept. It means that a function calls itself. This has the benefit of meaning that you can loop through data to reach a result.

The developer should be very careful with recursion as it can be quite easy to slip into writing a function which never terminates, or one that uses excess amounts of memory or processor power. However, when written correctly recursion can be a very efficient and mathematically-elegant approach to programming.

In this example, `tri_recursion()` is a function that we have defined to call itself ("recurse"). We use the `k` variable as the data, which decrements (`-1`) every time we recurse. The recursion ends when the condition is not greater than 0 (i.e. when it is 0).

To a new developer it can take some time to work out how exactly this works, best way to find out is by testing and modifying it.

```
def tri_recursion(k):
  if(k>0):
    result = k+tri_recursion(k-1)
    print(result)
  else:
    result = 0
  return result

print("\n\nRecursion Example Results")
tri_recursion(6)
```

Global and Local Variables in Python

Global variables are the one that are defined and declared outside a function and we need to use them inside a function.

```
# This function uses global variable s
def f():
    print s

# Global scope
s = "I love Python"
f()
```

If a variable with same name is defined inside the scope of function as well then it will print the value given inside the function only and not the global value.

```
# This function has a variable with
# name same as s.
def f():
    s = "Me too."
    print s

# Global scope
s = "I love Python"
f()
print s
```

The question is, what will happen, if we change the value of s inside of the function f()? Will it affect the global s as well? We test it in the following piece of code:

```
def f():
    print s

    # This program will NOT show error
    # if we comment below line.
    s = "Me too."

    print s

# Global scope
```

```
s = "I love Python"
f()
print s
```

To make the above program work, we need to use “global” keyword. We only need to use global keyword in a function if we want to do assignments / change them. global is not needed for printing and accessing. Why? Python “assumes” that we want a local variable due to the assignment to s inside of f(), so the first print statement throws this error message. Any variable which is changed or created inside of a function is local, if it hasn’t been declared as a global variable. To tell Python, that we want to use the global variable, we have to use the keyword **“global”**, as can be seen in the following example:

```
# This function modifies global variable 's'
def f():
   global s
   print s
   s = "Look for Python Variables Section"
   print s

# Global Scope
s = "Python is great!"
f()
print s
```

Now there is no ambiguity.

*args and **kwargs in python

*args and **kwargs magic variables

It is not necessary to write *args or **kwargs. Only the * (aesteric) is necessary. You could have also written *var and **vars. Writing *args and **kwargs is just a convention.

*args and **kwargs allow you to pass a variable number of arguments to a function. What does variable mean here is that you do not know beforehand that how many arguments can be passed to your function by the user so in this case you use these two keywords.

Usage of *args

*args is used to send a non-keyworded variable length argument list to the function. Let’s learn with example:

```
def test_var_args(normal_argument, *arg):
   print ("first normal arg:", normal_argument)
   for eachArg in arg:
      print ("eachArg through *arg :", eachArg)

test_var_args('h2kinfosys','Rishiz','Python','Course')
```

Usage of **kwargs

**kwargs allows you to pass keyworded variable length of arguments to a function. You should use **kwargs if you want to handle named arguments in a function. Here is an example to get you going with it:

```
def greet_me(**kwargs):
   if kwargs is not None:
      for key, value in kwargs.items():
         print(key, value)

greet_me(name="Rishi")
```

- *args = tuple of arguments - as positional arguments
- **kwargs = dictionary - whose keys become separate keyword arguments and the values become values of these arguments.

So if you want to use all three of these in functions then the order is:

```
some_func(norm_args,*args,**kwargs)
```

Anonymous functions:

In Python, anonymous function means that a function is without a name. As we already know that:

def keyword: is used to define the normal functions
lambda keyword: is used to create anonymous functions.

```
cube = lambda x: x*x*x

print(cube(5))
```

First Class functions in Python

Functions in Python are strangely different than any other language. Python supports the concept of First Class functions.

Properties of first-class functions:

- A function is an instance of the Object type.
- You can store the function in a variable.
- You can pass the function as a parameter to another function.
- You can return the function from a function.
- You can store them in data structures such as tuples, lists,

1. **Functions are Objects:**

```
# Functions are objects
def shout(text):
    return text.upper()

# You can assign function to another variable
lol = shout
print(lol("Hello"))
print(shout("Hello"))
print(type(shout)) # You can check the type of it - 'function'
```

2. Functions can be passed as arguments to other functions

Because functions are objects, we can pass them as arguments to other functions. Functions that can accept other functions as arguments are also called higher-order functions.

```
# Functions are objects
def shout(text):
    return text.upper()

# Functions can be passed as arguments to other functions
def greet(func):
    greeting = func("Hi, I am created by a function passed as
an argument.")
    print(greeting)

greet(shout)
```

3. Functions can return another function: Because functions are objects, we can return a function from another function. Below example also illustrates, how a function can have another function (called as local function – or nested functions)

```
def calculations(num, name):
    function_name = None

    def square():
        return num * num

    def add_them():
        return num+num

    if name == "square":
        print("you said square, here you go ")
        function_name = square
    else:
        print("adding two numbers")
        function_name = add_them

    return function_name
```

```
get_square = calculations(15, "square")
print(get_square())
```

Python Classes/Objects

What are classes and objects in Python?

Python is an object-oriented programming language. Unlike procedure-oriented programming, where the main emphasis is on functions, object-oriented programming stress on objects.

Object is simply a collection of data / state / properties (variables) and methods / behaviour (functions) that act on those data. And, **class is a blueprint for the object**.

We can think of class as a sketch (prototype) of a house. It contains all the details about the floors, doors, windows etc. Based on these descriptions we build the house. House is the object.

As, many houses can be made from a description, we can create many objects from a class. An object is also called an instance of a class and the process of creating this object is called **instantiation**.

Almost everything in Python is an object, with its properties and methods. A Class is like an object constructor, or a "blueprint" for creating objects.

Create a Class

To create a class, use the keyword class. Create a class named MyClass, with a property named x:

```
class MyClass:
  x = 5
```

A class creates a new local namespace where all its attributes are defined. Attributes may be data or functions.

Create Object

Now we can use the class named MyClass to create objects:

```
p1 = MyClass()
print(p1.x)
```

This will create a new instance object named p1. We can access attributes of objects using the object name prefix. Attributes may be data or method.

Constructors in Python

Class functions that begins with double underscore (__) are called special functions as they have special meaning.

Of one particular interest is the __init__() function. This special function gets called whenever a new object of that class is instantiated. All classes have a function called __init__(), which is always executed when the class is being initiated.

Use the __init__() function to **assign values to object properties**, or other operations that are necessary to do when the object is being created:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

p1 = Person("John", 36)

print(p1.name)
print(p1.age)
```

The __init__() function is called automatically every time the class is being used to create a new object.

Object Methods

Objects can also contain methods. Methods in objects are functions that belong to the object.

Let us create a method in the Person class:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

  def myfunc(self):
    print("Hello my name is " + self.name)

p1 = Person("John", 36)
p1.myfunc()
```

The self Parameter

The self parameter is a reference to the current instance of the class, and is used to access variables that belongs to the class.

```
class Person:
  def __init__(self, name, age):
    mysillyobject.name = name
    mysillyobject.age = age

  def myfunc(self, abc):
    print("Hello my name is " + abc.name)

p1 = Person("John", 36)
p1.myfunc()
```

Can you check type of this function?

Object Properties

You can modify properties on objects like this:

```
p1.age = 40
```

You can delete properties on objects by using the `del` keyword:

```
del p1.age
```

You can delete objects by using the `del` keyword:

```
del p1
```

Before moving ahead… lets understand advance functions.

Python Inheritance

Inheritance allows us to define a class that inherits all the methods and properties from another class.

Parent class is the class being inherited from, also called base class.

Child class is the class that inherits from another class, also called derived class.

Create a Parent Class

Any class can be a parent class, so the syntax is the same as creating any other class:

```
class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)

#Use the Person class to create an object, and then execute the printname method:

x = Person("John", "Doe")
x.printname()
```

Create a Child Class

To create a class that inherits the functionality from another class, send the parent class as a parameter when creating the child class:

```
class Student(Person):
  pass
```

Note: Use the pass keyword when you do not want to add any other properties or methods to the class

Use the Student class to create an object, and then execute the printname method:

```
x = Student("Mike", "Olsen")
x.printname()
```

Add the __init__() Function

So far we have created a child class that inherits the properties and methods from its parent.

We want to add the `__init__()` function to the child class (instead of the `pass` keyword).

The `__init__()` function is called automatically every time the class is being used to create a new object.

```
class Student(Person):
  def __init__(self, fname, lname):
    #add properties etc.
```

When you add the `__init__()` function, the child class will no longer inherit the parent's `__init__()`function.

Note: The child's `__init__()` function **overrides** the inheritance of the parent's `__init__()` function.

To keep the inheritance of the parent's `__init__()` function, add a call to the parent's `__init__()`function:

```
class Student(Person):
  def __init__(self, fname, lname):
          .__init__(self, fname, lname)
```

Now we have successfully added the __init__() function, and kept the inheritance of the parent class, and we are ready to add functionality in the `__init__()` function.

Add Properties

```
class Student(Person):
  def __init__(self, fname, lname):
    Person.__init__(self, fname, lname)
    self.graduationyear = 2019
```

In the example below, the year 2019 should be a variable, and passed into the Student class when creating student objects. To do so, add another parameter in the __init__() function:

```
class Student(Person):
  def __init__(self, fname, lname, year):
    Person.__init__(self, fname, lname)
    self.graduationyear = year

x = Student("Mike", "Olsen", 2019)
```

Add Methods

```
class Student(Person):
  def __init__(self, fname, lname, year):
    Person.__init__(self, fname, lname)
    self.graduationyear = year

  def welcome(self):
    print("Welcome", self.firstname, self.lastname, "to the class of", self.graduationyear)
```

Different forms of Inheritance:

1. Single inheritance: When a child class inherits from only one parent class, it is called as single inheritance. We saw an example above.

2. Multiple inheritance: When a child class inherits from multiple parent classes, it is called as multiple inheritance.
Unlike Java and like C++, Python supports multiple inheritance. We specify all parent classes as comma separated list in bracket.

```
# Python example to show working of multiple
# inheritance
class Base1(object):
    def __init__(self):
        self.str1 = "Geek1"
        print "Base1"

class Base2(object):
    def __init__(self):
        self.str2 = "Geek2"
        print "Base2"

class Derived(Base1, Base2):
    def __init__(self):
```

```
        # Calling constructors of Base1
        # and Base2 classes
        Base1.__init__(self)
        Base2.__init__(self)
        print "Derived"

    def printStrs(self):
        print(self.str1, self.str2)


ob = Derived()
ob.printStrs()
```

3. Multilevel inheritance: When we have child and grand child relationship.

```
# A Python program to demonstrate inheritance

# Base or Super class. Note object in bracket.
# (Generally, object is made ancestor of all classes)
# In Python 3.x "class Person" is
# equivalent to "class Person(object)"
class Base(object):

    # Constructor
    def __init__(self, name):
        self.name = name

    # To get name
    def getName(self):
        return self.name


# Inherited or Sub class (Note Person in bracket)
class Child(Base):

    # Constructor
    def __init__(self, name, age):
        Base.__init__(self, name)
        self.age = age

    # To get name
    def getAge(self):
        return self.age

# Inherited or Sub class (Note Person in bracket)
class GrandChild(Child):

    # Constructor
    def __init__(self, name, age, address):
        Child.__init__(self, name, age)
        self.address = address
```

```
    # To get address
    def getAddress(self):
        return self.address

# Driver code
g = GrandChild("Geek1", 23, "Noida")
print(g.getName(), g.getAge(), g.getAddress())
```

There are 2 built-in functions in Python that are related to inheritance. They are:

1. **isinstance**(): It checks the type of an object. Its syntax is:
isinstance(object_name, class_name) It would return True if the class of object_name is class_name else False.
For example:

```
# Python code to demonstrate issubclass()

print(isinstance(5, int))
```

The above code would show the following output:
True
This is because 5 is an integer and hence belongs to the class of int.
NOTE: 'int' is both a type and a class in Python.

2. issubclass(): It checks whether a specific class is the child class of another class or not. Its syntax is:
issubclass(childclass_name, parentclass_name)

It would return **True** if the entered child class is actually derived from the entered parent class else, it returns **False**.
For example:

```
# Python code to demonstrate issubclass()
class A():
    def __init__(self, a):
        self.a = a
class B(A):
    def __init__(self, a, b):
        self.b = b
        A.__init__(self, a)

print(issubclass(B, A))
```

Python Modules

Consider a module to be the same as a code library.

A file containing a set of functions you want to include in your application.

To create a module just save the code you want in a file with the file extension .py:

Save this code in a file named mymodule.py

```
def greeting(name):
  print("Hello, " + name)
```

Use a Module

Now we can use the module we just created, by using the import statement:

```
import packagename.mymodule

mymodule.greeting("Jonathan")
```

Note: When using a function from a module, use the syntax: *module_name.function_name*.

Variables in Module

The module can contain functions, as already described, but also variables of all types (arrays, dictionaries, objects etc):

```
person1 = {
  "name": "John",
  "age": 36,
  "country": "Norway"
}


import mymodule

a = mymodule.person1["age"]
print(a)
```

Naming a Module

You can name the module file whatever you like, but it must have the file extension .py

Re-naming a Module

You can create an alias when you import a module, by using the as keyword:

Create an alias for mymodule called mx:

```
import mymodule as mx

a = mx.person1["age"]
print(a)
```

Built-in Modules

There are several built-in modules in Python, which you can import whenever you like.

Import and use the platform module:

```
import platform

x = platform.system()
print(x)
```

Import From Module

You can choose to import only parts from a module, by using the from keyword.

The module named mymodule has one function and one dictionary:

```
def greeting(name):
  print("Hello, " + name)

person1 = {
  "name": "John",
  "age": 36,
  "country": "Norway"
}
```

Example

Import only the person1 dictionary from the module:

```
from mymodule import person1

print (person1["age"])
```