Functions

In your high-school, you must have learnt about functions in mathematics. A function is something that takes some input and provides some output, so there is some correspondence between the input and the output.

Functions in Programming Languages are similar. They take inputs and they provide outputs.

A function is sort of like a contract - imagine the function speaking to you - "If you give me this input, I will give you this output. If you give me a number, I will give you its square-root. If you give me the name of a person, I will tell you if he works in this company or not".

So what are the things we need when creating a function:

* We need a name for the function
* We need to have clearly defined inputs
* We need to have clearly defined outputs
* We finally need to have a definition - which is how the function already works

Here is an example of a function:

function square(x):

return x \* x

Name of the function is square It takes x as input and x should be a number It returns the square of x as output and that is a number too

You can then "call" the function as follows:

square(4)

square(10)

etc

Can I "call" a function within another function? Yes you can

Can I call the same function from within the function? Yes you can and they are called recursive functions

More

* <http://en.wikipedia.org/wiki/Procedural_programming>

# Libraries

Ok, now you know about functions and you look at your original code, and start grouping a bunch of statements together as functions.

And you run into a second issue soon... too many functions.

So you wonder... what do I do now?

The solution - group functions together into what are called libraries or modules.

Loop constructs (while and for) lead to iterative algorithm

**PARADIGMS**

Paradigms are ways to approach problem solving in a programming language.

Examples of Popular Paradigms:

* Procedural Programming
* Functional Programming
* Object Oriented Programming
* Aspect Oriented Programming

***Functional Programming***

First Class Functions

Concepts

1. Functions as First Class Objects

First Class Functions

What's a first class citizen?

Let us take "Numbers" as an example to understand the concept of First Class Citizens. Numbers are first class in all languages.

For something to classify as a first class citizen, it has to satisfy 3 conditions:

1. You should be able to assign it to a variable. Eg: x = 20
2. You should be able to pass it to a function. Eg: foo(20)
3. You should be able to return it from a function. Eg: return 20

We see that Numbers satisfy all these conditions, hence they are first class citizens.

In functional languages, functions are also first class. What does that mean?

Let us look at each condition separately:

Assigning to variables

1. You can assign it to a variable. We will take the example of a language like Python just to explain the concept. You don't need prior understanding of Python to understand these concepts. We choose Python because it has very simple syntax and is functional.

In a language like Python, what does it mean to assign something to a variable? Example:

x = y

What this means is, x is a reference to the same location as what y is referencing.

Ok, so what happens when you define a function:

>>> def foo():

... return 10

...

Defining a function creates an object in memory to hold the function definition. foo is then a variable that references this object. This is no different conceptually from what happens when you create any variable and initialize it with a value (eg: x = 20).

Now combining the above ideas, let us see what happens when we run the following piece of code:

>>> def foo():

... return 10

...

>>> x = foo

>>> type(x)

<type 'function'>

>>> x()

10

>>>

A function object gets created. foo is a variable that points to this function definition. When we say x = foo, we are just saying, let x be a second reference to the same object as what foo is referencing. foo is referencing a function, so also happens to reference the same function. If a variable references a function, you can call it. That's what we are doing next.

Please note that this is very different from saying:

x = foo()

The above notion exists in all languages including non-functional languages. Here we are saying, "call foo" and the return value of foo should be assigned to x. This is very different from x = foo.

Passing to functions

Once we get this basic idea of assigning functions to variables, the rest 2 conditions are easy to follow.

Consider this piece of code:

>>> def foo\_caller(foo):

... foo()

...

>>> def bar():

... print('Bar')

...

>>> foo\_caller(bar)

Bar

>>>

Let us understand this code, one step at a time. The first statement creates a function called foo\_caller. In other words, it creates an object that holds the function definition, and then makes foo\_caller a variable that references this function object.

Special note: Is foo created at this point in time? Remember that the function arguments don't get created unless the function is called, so no. foo is not created yet. foo is created only when foo\_caller is called. foo is created as many times as foo\_caller is called.

The next statement is similar. A function object gets created and bar is a variable that references it.

The next statement is magical: foo\_caller(bar). This means, execute the body of foo\_caller by passing a reference of bar to it.

Let us understand this better. Examine this simple function:

>>> def sum(x, y):

... return x + y

...

>>> a = 10

>>> b = 20

>>> sum(a, b)

30

>>>

What happens when we say sum(a, b). We are saying, execute the body of sum. Pass the references of a and b to it as arguments. The arguments are received as parameters x and y. It's as good as saying:

x = a

y = b

Isn't it?

And we know what that means! x = a means let x be a second reference to the object that a is referencing.

So now coming back to our foo\_caller function: foo\_caller(bar). We are saying, execute the body of foo\_caller and pass bar to it. This is received as the parameter foo. It's as good as saying:

foo = bar

Let foo be a second reference to what bar is referencing. We know what bar is referencing. It's a function. foo now becomes a second reference to the same function.

When we execute the body of foo\_caller, we are just asking it to execute the body of foo. This is nothing but the body of bar.

Returning functions from functions

This is the final case of first class functions.

Consider this piece of code:

>>> def foo():

... def bar():

... print('Bar')

... return bar

...

>>> x = foo()

>>> x()

Bar

>>>

We already know most of it. But let's look at them one statement at a time.

After executing the first statement, the function object gets created and foo is a reference to it. Remember that bar is not yet created. bar is created only if foo is called. The next statement is saying, call foo. We execute the body of foo and the return value of foo is assigned to x. What happens when we execute the body of foo? A function object gets created and bar is a reference to it. This reference is then returned. This returned reference is received as x.

This is similar to the following:

>>> def get\_x():

... x = 10

... return x

...

>>> y = get\_x()

>>> y

10

>>>

See that? Return value of get\_x is assigned to y. That's as good as saying:

y = x

Similarly, the above is doing:

x = bar

We know what that means! x becomes a second reference to the object that bar is referencing. bar is referencing a function, so x happens to reference the same function. So when x is called, it's just calling the same function as bar. That prints "Bar".

Read more about functions as first class objects here: <https://en.wikipedia.org/wiki/First-class_function>

2. Data Transforms in Functional Programming Languages

3. Higher Order Functions

4. Lambda Functions