Python Basic - Part 2

1. 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. These parameters will be call arguments when use as variable in function scope.

In Python a function is defined using the `def` keyword

```
def first_function():
        print("Hello World!")

first_function()
# -> Hello World!
```

> **Note**
>
> The code in the block that follows the `def` statement is the body of the function. This code is executed when the function is called, not when the function is first defined.

The `first_function()` lines after the function are function calls. In code, a function call is just the function's name followed by parantheses, possibly with some number of arguments in between the parenthese.

def Statements with Parameters

When you call the `print()` function, you pass them values, called arguments, by typing them between the parentheses. You can also define your own functions that accept arguments.

```
def say_hello(name):
        print(f"Hello {name}!")

say_hello("Sam")
# -> Hello Sam!
```

Return Values and Return Statements

When creating a function using the def statement, you can specify what the return value should be with a return statement. A return statement consists of the following:

- The return keyword
- The value or expression that the function should return

When an expression is used with a return statement, the return value is what this expression evaluates to

```
def sum_number(x,y):
        return x+y

result = sum_number(10,20)

print(result)

# -> 30
```

The None Value

In Python, there is a value called None, which represents the absence of a value. The None value is the only value of the NoneType data type. (Other programming languages might call this value null, nil, or undefined.) Just like the Boolean True and False values, None must be typed with a capital N.

```
hello = print("Hello!")

None == hello
# -> True
```

Local and Global Scope

Parameters and variables that are assigned in a called function are said to exist in that function's local scope. Variables that are assigned outside all functions are said to exist in the global scope. A variable that exists in a local scope is called a local variable, while a variable that exists in the global scope is called a global variable. A variable must be one or the other; it cannot be both local and global

Think of a scope as a container for variables. When a scope is destroyed, all the values stored in the scope's variables are forgotten. There is only one global scope, and it is created when your program begins. When your program terminates, the global scope is destroyed, and all its variables are forgotten. Otherwise, the next time you ran a program, the variables would remember their values from the last time you ran it.

A local scope is created whenever a function is called. Any variables assigned in the function exist within the function's local scope. When the function returns, the local scope is destroyed, and these variables are forgotten. The next time you call the function, the local variables will not remember the values stored in them from the last time the function was called. Local variables are also stored in frame objects on the call stack.

Scopes matter for several reasons:

- Code in the global scope, outside of all functions, cannot use any local variables.
- However, code in a local scope can access global variables.
- Code in a function's local scope cannot use variables in any other local scope.

- You can use the same name for different variables if they are in different scopes. That is, there can be a local variable named spam and a global variable also named spam.

Local Variables cannot be used in the Global Scope

```
def spam():
        eggs = 3.14

spam()
print(eggs)
```

The program will output an error because `eggs` variable is no long available

Error

```
Traceback (most recent call last):
 File "C:/test1.py", line 4, in
   print(eggs)
NameError: name 'eggs' is not defined
```

The error happens because the eggs variable exists only in the local scope created when spam() is called. Once the program execution returns from spam, that local scope is destroyed, and there is no longer a variable named eggs. So when your program tries to run print(eggs), Python gives you an error saying that eggs is not defined. This makes sense if you think about it; when the program execution is in the global scope, no local scopes exist, so there can't be any local variables. This is why only global variables can be used in the global scope.

Local Scopes cannot use variables in other Local Scopes

A new local scope is created whenever a function is called, including when a function is called from another function. Consider this program:

```
def spam():
    eggs = 99
    bacon()
    print(eggs)

   def bacon():
    ham = 101
    eggs = 0

spam()
```

When the program starts, the spam() function is called, and a local scope is created. The local variable eggs is set to 99. Then the bacon() function is called, and a second local scope is created. Multiple local scopes can exist at the same time. In this new local scope, the local variable ham is set to 101, and a local variable eggs—which is different from the one in spam()'s local scope—is also created and set to 0.

When bacon() returns, the local scope for that call is destroyed, including its eggs variable. The program execution continues in the spam() function to print the value of eggs. Since the local scope for the call to spam() still exists, the only eggs variable is the spam() function's eggs variable, which was set to 99. This is what the program prints.

The upshot is that local variables in one function are completely separate from the local variables in another function

Global Variables can be read from a Local Scope

```
def spam():
    print(eggs)
eggs = 42
spam()
print(eggs)
```

Since there is no parameter named eggs or any code that assigns eggs a value in the spam() function, when eggs is used in spam(), Python considers it a reference to the global variable eggs. This is why 42 is printed when the previous program is run.

Local and Global Variable with the same Name

Technically, it's perfectly acceptable to use the same variable name for a global variable and local variables in different scopes in Python. But, to simplify your life, avoid doing this.

```
def spam():
    eggs = 'spam local'
    print(eggs)    # prints 'spam local'

def bacon():
    eggs = 'bacon local'
    print(eggs)    # prints 'bacon local'
    spam()
    print(eggs)    # prints 'bacon local'

eggs = 'global'
bacon()
print(eggs)        # prints 'global'
```

There are actually three different variables in this program, but confusingly they are all named eggs. The variables are as follows:

1. A variable named eggs that exists in a local scope when spam() is called.
2. A variable named eggs that exists in a local scope when bacon() is called.
3. A variable named eggs that exists in the global scope.

Since these three separate variables all have the same name, it can be confusing to keep track of which one is being used at any given time. This is why you should avoid using the same variable name in different scopes.

The global Statement

If you need to modify a global variable from within a function, use the global statement. If you have a line such as global eggs at the top of a function, it tells Python, “In this function, eggs refers to the global variable, so don’t create a local variable with this name.”

```
def spam():
  global eggs
  eggs = 'spam'

eggs = 'global'
spam()
print(eggs)

# -> spam
```

Because eggs is declared global at the top of spam(), when eggs is set to 'spam', this assignment is done to the globally scoped eggs. No local eggs variable is created.

There are four rules to tell whether a variable is in a local scope or global scope:

- If a variable is being used in the global scope (that is, outside of all functions), then it is always a global variable.
- If there is a global statement for that variable in a function, it is a global variable.
- Otherwise, if the variable is used in an assignment statement in the function, it is a local variable.
- But if the variable is not used in an assignment statement, it is a global variable.

2. Python Built-in Method

In Python, there are many functions that already built by Python developer as well as contribution from the community around the world. These functions sometime can be call **method** which in a sense allow you to call and pass parameter just like function. Each data type in Python has it own method that can be use to transform, or convert provided value into something else

Below are methods that most commonly use in Python

STR - STRING METHOD

capitalize()

The `capitalize()` method returns a string where the first character is upper case, and the rest is lower case.

Syntax

```
_string_.capitalize()
```

Parameter Value

`No parameter`

Example

```
txt = "python is FUN!"
x = txt.capitalize()
print(x)

# PYTHON IS FUN!
```

isalpha()

The `isalpha()` method returns True if all the characters are alphabet letters (a-z).
Example of characters that are not alphabet letters: (space)!#%&? etc.

Syntax

```
_string_.isalpha()
```

Parameter Value

`No parameter`

Example

```
txt = "Company10"
x = txt.isalpha()
print(x)

# False
```

lower()

The `lower()` method returns a string where all characters are lower case.
Symbols and Numbers are ignored.

Syntax

```
_string_.lower()
```

Parameter Value

`No parameter`

Example

```
txt = "python is FUN!"
x = txt.capitalize()
print(x)

# python is fun!
```

More str method please visit w3school

3. IF Statement

Python supports the usual logical conditions from mathematics:

- Equals: a == b
- Not Equals: a != b
- Less than: a < b
- Less than or equal to: a <= b
- Greater than: a > b
- Greater than or equal to: a >= b

These conditions can be used in several ways, most commonly in "if statements" and loops.
An "if statement" is written by using the if keyword.

```
a = 33
b = 200
if b > a:
  print("b is greater than a")

# b is greater than a
```

In this example we use two variables, a and b, which are used as part of the if statement to test whether b is greater than a.
As a is 33, and b is 200, we know that 200 is greater than 33, and so we print to screen that "b is greater than a".

Indentation

Python relies on indentation (whitespace at the beginning of a line) to define scope in the code. Other programming languages often use curly-brackets for this purpose.

```
a = 33
b = 200
```

```
if b > a:
print("b is greater than a") # you will get an error, IndentationError
```

Elif

The elif keyword is pythons way of saying "if the previous conditions were not true, then try this condition".

```
a = 33
b = 33
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")

# a and b are equal
```

In this example a is equal to b, so the first condition is not true, but the elif condition is true, so we print to screen that "a and b are equal".

Else

The else keyword catches anything which isn't caught by the preceding conditions.

```
a = 200
b = 33
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
else:
  print("a is greater than b")
```

```
# a is greater than b
```

In this example a is greater than b, so the first condition is not true, also the elif condition is not true, so we go to the else condition and print to screen that "a is greater than b".

You can also have an `else` without the `elif`:

```
a = 200
b = 33
if b > a:
  print("b is greater than a")
else:
  print("b is not greater than a")

# b is not greater than a
```

And

The and keyword is a logical operator, and is used to combine conditional statements:

```
a = 200
b = 33
c = 500
if a > b and c > a:
  print("Both conditions are True")

# Both conditions are True
```

Or

The `or` keyword is a logical operator, and is used to combine conditional statements:

```
a = 200
b = 33
c = 500
if a > b or a > c:
  print("At least one of the conditions is True")

# At least one of the conditions is True
```

Nested If

You can have `if` statements inside `if` statements, this is called *nested* `if` statements.

```
x = 41

if x > 10:
  print("Above ten,")
  if x > 20:
    print("and also above 20!")
  else:
    print("but not above 20.")

# Above ten,
# and also above 20!
```