6.1-Functions Basic

KeytoDataScience.com

Table of Contents

💡💡💡💡💡 **Functions** 💡💡💡💡💡

- **Function is a group of related statements that performs a specific task.**
- **Functions help break our program into smaller and modular chunks. As our program grows larger and larger, functions make it more readable, organized and manageable.**
- **It avoids repetition and makes the code reusable.**

1 Syntax of Function

1.1 Using Functions

Functions are groups of code that have a name, and can be called using parentheses.

We've seen functions before. For example, `print` in Python 3 is a function:

In [42]:
```
print('KeytoDataScience')
```

```
KeytoDataScience
```

Here `print` is the function name, and `'KeytoDataScience'` is the function's *argument*.

In addition to arguments, there are *keyword arguments* that are specified by name.

One available keyword argument for the `print()` function (in Python 3) is `sep` **(as seen in the image below)**, which tells what character or characters should be used to separate multiple items:

> **Tip:** Press $SHIFT + TAB$ by placing curser inside print(**place curser here**)

```
print('KeytoDataScience')
```

```
Docstring:
print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

Prints the values to a stream, or to sys.stdout by default.
```

In [43]:
```
print(1, 2, 3)
```

```
1 2 3
```

In [44]:
```
print(1, 2, 3, sep='--')
```

```
1--2--3
```

1.2 Defining Functions

Functions become even more useful when we begin to define our own, organizing functionality to be used in multiple places. In Python, functions are defined with the `def` statement.

In [1]:
```
def function_name(parameters):
    """docstring"""
    statement(s)
```

```
-Keyword 'def' that marks the start of the function header.
-A function name to uniquely identify the function.
-Parameters (arguments) through which we pass values to a function. They
are optional.
-A colon (:) to mark the end of the function header.
-Optional documentation string (docstring) to describe what the function
does.
-One or more valid python statements that make up the function body.
Statements must have the same indentation level (usually 4 spaces).
-An optional return statement to return a value from the function.
```

For example, we can create a function to print `hello`

In [2]:
```
# Example
def say_hello():
```

```
    """Simple function that prints Hello"""
    print("Hello")
```

In [3]:
```
say_hello
```

Out[3]:
```
<function __main__.say_hello()>
```

In [4]:
```
help(say_hello)
```

```
Help on function say_hello in module __main__:

say_hello()
    Simple function that prints Hello
```

In [5]:
```
say_hello()
```

```
Hello
```

1.3 Function with arguments

In [6]:
```
def greeting(name):
    print("Hello from {}".format(name))
```

In [7]:
```
greeting("KeytoDataScience")
```

```
Hello Data Lab's
```

1.4 Function with return statements

In [8]:
```
def add(x,y):
    return x+y
```

In [9]:
```
add(2,4)
```

Out[9]:
```
6
```

In [10]:
```
def check(x):
    if x > 0:
        return "Positive Number"
    elif x < 0:
        return "Negative Number"
    return 'Number is 0'
```

In [11]:
```
check(3)
```

Out[11]:
```
'Positive Number'
```

1.5 `*args` and `**kwargs`: Flexible Arguments

Function with variable number of arguments

Sometimes you might wish to write a function in which you don't initially know how many arguments the user will pass. In this case, you can use the special form `*args` and `**kwargs` to catch all arguments that are passed. Here is an example:

In [12]:
```
#*args and **kwargs allow us to pass a variable number of arguments to a function

def add_v(*args):
    for i in args:
        print(i)
```

In [13]:
```
add_v(1,2,3,4,5)
```

```
1
2
3
4
5
```

In [14]:
```
def concatenate(**kwargs):
    result = ""
    for arg in kwargs.values():
        result += arg
    return result
```

In [15]:
```
print(concatenate(a="Python ", b="Functions", e="!"))
```

```
Python Functions!
```

In [45]:
```
def catch_all(*args, **kwargs):
    print("args =", args)
    print("kwargs = ", kwargs)
```

In [ ]:
```
catch_all(1, 2, 3, a=4, b=5)
```

In [ ]:
```
catch_all('a', keyword=2)
```

Here it is not the names `args` and `kwargs` that are important, but the `*` characters preceding them. `args` and `kwargs` are just the variable names often used by convention, short for "arguments" and "keyword arguments". The operative difference is the asterisk characters: a single `*` before a variable means "expand this as a sequence", while a double `**` before a variable means "expand this as a dictionary". In fact, this syntax can be used not only with the function definition, but with the function call as well!

In [ ]:
```
inputs = (1, 2, 3)
```

```
keywords = {'pi': 3.14}

catch_all(*inputs, **keywords)
```

↥ back to top

2 Functions are First Class objects

-We Can store the function in a variable.
-We Can pass the function as a parameter to another function.
-We Can return the function from a function.
-Functions can be stored in data structures such as lists.

2.1 Storing functions in a variable

In [16]:
```
def mul(x,y):
    return x*y
```

In [17]:
```
result = mul(2,9)
```

In [18]:
```
result
```

Out[18]:
```
18
```

In [19]:
```
x  = mul
```

In [20]:
```
x
```

Out[20]:
```
<function __main__.mul(x, y)>
```

In [21]:
```
x(2,8)
```

Out[21]:
```
16
```

2.2 Storing function in a List

In [22]:
```
l = [mul, 1, 2, 3]
```

In [23]:
```
l[0]
```

Out[23]:
```
<function __main__.mul(x, y)>
```

In [24]:
```
l[0](2,9)
```

```
18
```

Out[24]:

2.3 Passing function as a argument to another function

In [25]:
```
def welcome():
    print("This is a welcome function")
```

In [26]:
```
def greet(func):
    print("This is a greet function")
    func()
```

In [27]:
```
# Passing Functiona
greet(welcome)
```

```
This is a greet function
This is a welcome function
```

↥ back to top

3 Nested Function

In [28]:
```
def speak_func(text, volume):
    def whisper():
        return text.lower() + '...'
    def yell():
        return text.upper() + '!'
    if volume < 5:
        return whisper()
    else:
        return yell()
```

In [29]:
```
speak_func("Hello", 7)
```

Out[29]:
```
'HELLO!'
```

3.1 Closure

A function which is defined inside another function is known as nested function. Nested functions are able to access variables of the enclosing scope.

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

In [30]:
```
def print_msg(msg):
    # This is the outer enclosing function

    def printer():
        # This is the nested function
        print(msg)
```

```
    return printer
```

In [31]:
```
another = print_msg("Hello")
another()
```

```
Hello
```

In [32]:
```
del print_msg
```

In [33]:
```
print_msg
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_10016/1275571278.py in <module>
----> 1 print_msg

NameError: name 'print_msg' is not defined
```

In [34]:
```
another
```

Out[34]:
```
<function __main__.print_msg.<locals>.printer()>
```

In [35]:
```
another()
```

```
Hello
```

3.2 Decorators

Decorators can be thought of as functions which modify the functionality of another function. They help to make your code shorter and more **"Pythonic"**.

In [36]:
```
#@my_decorator
def say_hi():
    return "Hi"
```

In [37]:
```
def my_decorator(func):
    def wrapper():
        print("Changin the result to UpperCase")
        print(func().upper())
    return wrapper
```

In [38]:
```
say_hi()
```

Out[38]:
```
'Hi'
```

In [39]:
```
decorated = my_decorator(say_hi)
```

In [40]:
```
decorated()
```

```
Changin the result to UpperCase
HI
```

↥ back to top

Great Job!
