Data Science

Introduction to Python

CS202_03.5

Functions

Creating functions

We define a function by using the “def” keyword

```
def my_func():
  print("Hello World!”)
```

After we write a function we would need to call it to use it. We use the name of the function to call the function.

```
my_function()
```

Functions

Arguments

Information can be passed into functions as arguments.

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

The following example has a function with one argument (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_func(name):
  print(name + " is awesome")

my_function("Alice") #this will return “Alice is awesome”
my_function("Rachel")
my_function(“Colette")
```

We also call arguments parameters

Functions

Doc Strings and commenting

Its always a good idea to comment your code, especially in functions.
We can have doc strings, a standard habit one should have is to write parameters and return values of functions:

```
def my_func(name):
"""parameters: <str> name of person
Return: <str> a statement about the person """
  print(name + " is awesome ")
```

We use triple quotes for doc strings and we can use doc strings in the following way:

```
my_func(name).__doc__
```

This will print out your doc string, so if you forget what your function returns or what your function takes in as parameters you can use doc strings to recall.

Otherwise, use comments with a “#”. But this is for comments and are usually not used in this way. We can look at another example of what is a standard way to comment.

```
def my_func(name):
  #parameters: <str> name of person
  #return: <str> a statement about the person
  print(name + " is awesome “)
```

Functions

Number of Arguments

A function must be called with the correct number of arguments. Meaning that if your function expects 2 arguments, you have to call the function with 2 arguments, not more, and not less.

```
def my_function(fname, lname):
  print(fname + " " + lname)

my_function("Alice", “Kwon")
```

If you try to call the function with 1 or 3 arguments, you will get an error:

```
my_function("Alice", “Kwon”)
```

Functions

Arbitrary Arguments *args

If you do not know how many arguments that will be passed into your function, add a * before the parameter name in the function definition.

This way the function will receive a *tuple* of arguments, and can access the items accordingly:

```
def my_function(*kids):
  print("The youngest child is " + kids[2])

my_function("Alice", "Rachel", "Colette")
```

This will return “The youngest child is Colette”

Functions

Keyword Arguments

You can also send arguments with the *key* = *value* syntax.

This way the order of the arguments does not matter.

```
def my_function(child3, child2, child1):
  print("The youngest child is " + child3)

my_function(child1 = "Alice", child2 = "Rachel", child3 = “Colette")
```

Arbitrary Keyword Arguments, **kwargs

If you do not know how many keyword arguments that will be passed into your function, add two asterisk: ** before the parameter name in the function definition.

This way the function will receive a *dictionary* of arguments, and can access the items accordingly:

```
def my_function(**kid):
  print("His last name is " + kid["lname"])

my_function(fname = "Tobias", lname = "Refsnes")
```

Functions

You can have default parameter values:

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

my_function("Sweden")
my_function("India")
my_function()
my_function(“Brazil”)
```

You can send any data types of argument 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 an argument, 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)
```

Functions

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))
```

Most of the time you will use the “return” statement.
You can use the “pass” statement just as you did for for loops.

```
def myfunction():
  pass
```

Functions

Recursion

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

Recursion 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

tri_recursion(6)
```

Functions

Lambda

A lambda function is a small anonymous function.

A lambda function can take any number of arguments, but can only have one expression.

The syntax is; lambda <arguments> : *expression*

```
x = lambda a : a + 10
print(x(5))
```

The above lambda function adds 10 to the argument a.

Why would one use lambda functions?

The power of lambda is better shown when you use them as an anonymous function inside another function.

Say you have a function definition that takes one argument, and that argument will be multiplied with an unknown number:

```
def myfunc(n):
  return lambda a : a * n

mydoubler = myfunc(2)

print(mydoubler(11))
```

The above should return 22. We will see that when manipulating data frames, it is often very handy to use lambda functions on its own.