Functions & Conditionals & Loops

21.09.2019

Python Notes

FUNCTIONS

Intro

- 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.

Creating a Function

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

```
Example
def sayHey():
  print("Hey!")
```

Call a Function

- To call a function, use the function name followed by parenthesis:

```
Example
def sayHey():
  print("Hey!")

sayHey()
```

Parameters

- You can add as many parameters as you want, just separate them with a comma.

```
Example
def my_function(name):
  print(name + " Yildiz")

my_function("Erdogan")
```

Default Parameter Value

- If we call the function without parameter, it uses the default value:

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

my_function("Germany")  # returns I am from Germany
my_function() # returns I am from Turkey
```

Return Values

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

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

print(my_function(3)) # prints 15
```

Keyword Arguments

- You can also send arguments with the *key* = *value* syntax.
- This way the order of the arguments does not matter.
- The phrase *Keyword Arguments* are often shortened to *kwargs* in Python documentations.

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

my_function(child1 = "Emil", child2 = "Tobias", child3 = "Linus")
```

Arbitrary Arguments

- 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:

```
Example: If the number of arguments are unknown, add a * before the parameter name:
def my_function(*kids):
  print("The youngest child is " + kids[2])

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

```
Example: Flexible functions
def adder(*num): #entered parameters are stored in num and can be reached by num[index]
    sum = 0

    for n in num:
        sum = sum + n
    print("Sum:",sum)

adder(3,5)
adder(4,5,6,7)
adder(1,2,3,5,6)
```

Recursion

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

```
Example:
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)
```

LAMBDA FUNCTIONS

Intro

- A lambda function is a small anonymous function.
- A lambda function can take any number of arguments, but can only have one expression.

Syntax

- `lambda` ***`arguments`*** `:` ***`expression`***

```
Example:
func = lambda a, b : a * b
print(func(5, 6))
```

Why 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:

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

- Use that function definition to make a function that always doubles the number you send in:

- Or, use the same function definition to make a function that always *triples* the number you send in:

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

mydoubler = myfunc(2) #mydoubler(a) becomes another function: it returns a*2
mytripler = myfunc(3) #mytripler(a) becomes another function: it returns a*3

print(mydoubler(11)) #prints 22
print(mytripler(11)) #prints 33
```

CONDITIONALS

If Statements

- An "if statement" is written by using the if keyword.

```
Example:
a = 10
b = 15
if b > a:
  print("b is greater than a")
```

Identation

- 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.

Elif

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

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

Else

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

```
Example:
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")
```

Nested If

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

```
Example:
x = 41

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

WHILE LOOPS

Intro

- With the **while** loop we can execute a set of statements as long as a condition is true.

```
Example: Print i as long as i is less than 6:
i = 1
while i < 6:
  print(i)
  i += 1
```

Break Statement

- With the break statement we can stop the loop even if the while condition is true:

```
Example: Exit the loop when i is 3:i = 1
while i < 6:
  print(i)
  if i == 3:
    break
  i += 1
```

Continue Statement

- With the continue statement we can stop the current iteration, and continue with the next:
- With the break statement we can stop the loop even if the while condition is true:

```
Example: Continue to the next iteration if i is 3:
i = 0
while i < 6:
  i += 1
  if i == 3:
    continue
  print(i)
```

FOR LOOPS

Intro

- A for loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).
- This is less like the for keyword in other programming languages, and works more like an iterator method as found in other object-orientated programming languages.
- With the for loop we can execute a set of statements, once for each item in a list, tuple, set etc.

Example: Print each fruit in a fruit list:

```
fruits = ["apple", "banana", "cherry"]
for key in fruits:
  print(key)
```

Looping Through a String

- Even strings are iterable objects, they contain a sequence of characters:

Example: Loop through the letters in the word "banana":

```
for x in "banana":
  print(x)
```

The Range Function

- To loop through a set of code a specified number of times, we can use the **range()** function,
- The **range()** function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.

Example: Using the range() function:

```
for x in range(6):
  print(x) # prints 0,1,2,3,4,5
```

- Note that range(6) is not the values of 0 to 6, but the values 0 to 5.

Example: Using the start parameter:

```
for x in range(2,6):
  print(x) # prints 2,3,4,5
```

Example: Increment the sequence with 3

```
for x in range(2,30,3):
  print(x) # prints 2,5,8,11,14,17,20,23,26,29
```

Nested Loops

Example: Print each adjective for every fruit

```
adj = ["red", "big", "tasty"]
fruits = ["apple", "banana", "cherry"]

for x in adj:
  for y in fruits:
    print(x, y)
```

More blogs