codecademy

Functions

Lambda Functions

Lambda Functions

A lambda function in Python is a simple, anonymous function that is defined without a name. Lambda functions are useful when we want to write a quick function in one line that can be combined with other built-in functions such as `map()`, `filter()`, and `apply()`. This is the syntax to define lambda functions:

```
lambda argument(s): expression
```

Function Parameters

Sometimes functions require input to provide data for their code. This input is defined using *parameters*.
Parameters are variables that are defined in the function definition. They are assigned the values which were passed as arguments when the function was called, elsewhere in the code.
For example, the function definition defines parameters for a character, a setting, and a skill, which are used as inputs to write the first sentence of a book.

```
def write_a_book(character, setting, special_skill):
  print(character + " is in " +
        setting + " practicing her " +
        special_skill)
```

Multiple Parameters

Python functions can have multiple *parameters*. Just as you wouldn't go to school without both a backpack and a pencil case, functions may also need more than one input to carry out their operations.
To define a function with multiple parameters, parameter names are placed one after another, separated by commas, within the parentheses of the function definition.

```
def ready_for_school(backpack, pencil_case):
  if (backpack == 'full' and pencil_case == 'full'):
    print ("I'm ready for school!")
```

Functions

Some tasks need to be performed multiple times within a program. Rather than rewrite the same code in multiple places, a function may be defined using the `def` keyword. Function definitions may include parameters, providing data input to the function. Functions may return a value using the `return` keyword followed by the value to return.

```
# Define a function my_function() with parameter x

def my_function(x):
  return x + 1

# Invoke the function

print(my_function(2))      # Output: 3
print(my_function(3 + 5))  # Output: 9
```

Function Indentation

Python uses indentation to identify blocks of code. Code within the same block should be indented at the same level. A Python function is one type of code block. All code under a function declaration should be indented to identify it as part of the function. There can be additional indentation within a function to handle other statements such as `for` and `if` so long as the lines are not indented less than the first line of the function code.

```
# Indentation is used to identify code blocks

def testfunction(number):
  # This code is part of testfunction
  print("Inside the testfunction")
  sum = 0
  for x in range(number):
    # More indentation because 'for' has a code block
    # but still part of he function
    sum += x
  return sum
print("This is not part of testfunction")
```

Calling Functions

Python uses simple syntax to use, invoke, or *call* a preexisting function. A function can be called by writing the name of it, followed by parentheses.
For example, the code provided would call the `doHomework()` method.

```
doHomework()
```

Function Arguments

Parameters in python are variables — placeholders for the actual values the function needs. When the function is *called*, these values are passed in as *arguments*.
For example, the arguments passed into the function `.sales()` are the "The Farmer's Market", "toothpaste", and "$1" which correspond to the parameters `grocery_store`, `item_on_sale`, and `cost`.

```
def sales(grocery_store, item_on_sale, cost):
  print(grocery_store + " is selling " + item_on_sale + " for
" + cost)

sales("The Farmer's Market", "toothpaste", "$1")
```

Function Keyword Arguments

Python functions can be defined with named arguments which may have default values provided. When function arguments are passed using their names, they are referred to as keyword arguments. The use of keyword arguments when calling a function allows the arguments to be passed in any order — *not* just the order that they were defined in the function. If the function is invoked without a value for a specific argument, the default value will be used.

```
def findvolume(length=1, width=1, depth=1):
  print("Length = " + str(length))
  print("Width = " + str(width))
  print("Depth = " + str(depth))
  return length * width * depth;

findvolume(1, 2, 3)
findvolume(length=5, depth=2, width=4)
findvolume(2, depth=3, width=4)
```

Returning Multiple Values

Python functions are able to return multiple values using one `return` statement. All values that should be returned are listed after the `return` keyword and are separated by commas.

In the example, the function `square_point()` returns `x_squared`, `y_squared`, and `z_squared`.

```
def square_point(x, y, z):
  x_squared = x * x
  y_squared = y * y
  z_squared = z * z
  # Return all three values:
  return x_squared, y_squared, z_squared

three_squared, four_squared, five_squared = square_point(3,
4, 5)
```

The Scope of Variables

In Python, a variable defined inside a function is called a local variable. It cannot be used outside of the scope of the function, and attempting to do so without defining the variable outside of the function will cause an error.

In the example, the variable `a` is defined both inside and outside of the function. When the function `f1()` is implemented, `a` is printed as `2` because it is locally defined to be so. However, when printing `a` outside of the function, `a` is printed as `5` because it is implemented outside of the scope of the function.

```
a = 5

def f1():
  a = 2
  print(a)

print(a)   # Will print 5
f1()       # Will print 2
```

Returning Value from Function

A `return` keyword is used to return a value from a Python function. The value returned from a function can be assigned to a variable which can then be used in the program.
In the example, the function `check_leap_year` returns a string which indicates if the passed parameter is a leap year or not.

```
def check_leap_year(year):
  if year % 4 == 0:
    return str(year) + " is a leap year."
  else:
    return str(year) + " is not a leap year."

year_to_check = 2018
returned_value = check_leap_year(year_to_check)
print(returned_value) # 2018 is not a leap year.
```

Global Variables

A variable that is defined outside of a function is called a global variable. It can be accessed inside the body of a function.
In the example, the variable `a` is a global variable because it is defined outside of the function `prints_a`. It is therefore accessible to `prints_a`, which will print the value of `a`.

```
a = "Hello"

def prints_a():
  print(a)

# will print "Hello"
prints_a()
```

Parameters as Local Variables

Function parameters behave identically to a function's local variables. They are initialized with the values passed into the function when it was called.
Like local variables, parameters cannot be referenced from outside the scope of the function.
In the example, the parameter `value` is defined as part of the definition of `my_function`, and therefore can only be accessed within `my_function`. Attempting to print the contents of `value` from outside the function causes an error.

```
def my_function(value):
  print(value)

# Pass the value 7 into the function
my_function(7)

# Causes an error as `value` no longer exists
print(value)
```

Print Share