Unit 3 -Functions, Scoping and Abstraction

Definition

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.

Functions are a way of creating computational elements that we can think of as primitives. They provide decomposition and abstraction.

Decomposition creates structure. It allows us to break a program into parts that are reasonably self-contained and that may be reused in different settings.

Abstraction hides detail. It allows us to use a piece of code as if it were a black box—that is, something whose interior details we cannot see, don't need to see, and shouldn't even want to see.31 The essence of abstraction is preserving information that is relevant in a given context, and forgetting information that is irrelevant in that context. The key to using abstraction effectively in programming is finding a notion of relevance that is appropriate for both the builder of an abstraction and the potential clients of the abstraction. That is the true art of programming.

Creating a function

In [ ]:
```
def my_function():
  print("Hello from a function")
```

Calling a Function

In [ ]:
```
my_function()
```

```
Hello from a function
```

Parameters/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:

In [ ]:
```
def my_function(fname):
  print('Hello,' ,fname)

my_function("John")
my_function("Luke")
my_function("Mark")
```

```
Hello, John
Hello, Luke
Hello, Mark
```

By default, 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.

In [ ]:
```
def my_function(fname, lname):
  print(fname, lname)

my_function("Emily", "Bronte")
```

```
Emily Bronte
```

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:

In [ ]:
```
def my_function(*students):
  print("Hi", students[2])

my_function('Mark', 'Luke', 'John')
```

```
Hi John
```

Keyword Arguments

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

This way the order of the arguments does not matter.

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

my_function(child1 = "Mark", child2 = "Luke", child3 = "John")
```

```
The youngest child is John
```

Default Arguments

The following example shows how to use a default parameter value.

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

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

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

```
I am from Sweden
I am from India
I am from Norway
I am from Brazil
```

Arbitrary Keyword Arguments

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:

If the number of keyword arguments is unknown, add a double ** before the parameter name:

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

my_function(fname = "Mark", lname = "Timothy")
```

```
His last name is Timothy
```

Mixing Different Types of Arguments

args and kwargs can be used in the same function once:

In [ ]:
```
def function(*args, **kwargs):
  print(args)
  print(kwargs)

function('abc', 'def', 'xyz', x='abc', y='def')
```

```
('abc', 'def', 'xyz')
{'x': 'abc', 'y': 'def'}
```

In the function below, abc is given to p1

In [ ]:
```
def function(p1, *args, **kwargs):
  print(p1)
  print(args)
  print(kwargs)

function('abc', 'def', 'xyz', x='abc', y='def')
```

```
abc
('def', 'xyz')
{'x': 'abc', 'y': 'def'}
```

The function below gives an error because there is no way to defining which arguments are for args and which one is for p1

In [ ]:
```
def function(*args, p1, **kwargs):
  print(p1)
  print(args)
  print(kwargs)

function('abc', 'def', 'xyz', x='abc', y='def')
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-49-0cfcd52d39a6> in <module>
      4   print(kwargs)
      5
----> 6 function('abc', 'def', 'xyz', x='abc', y='def')

TypeError: function() missing 1 required keyword-only argument: 'p1'
```

You can however pass p1 as a keyword argument as shown below:

In [ ]:
```
def function(*args, p1, **kwargs):
  print(p1)
  print(args)
  print(kwargs)

function('abc', 'def', p1='xyz', x='abc', y='def')
```

```
xyz
('abc', 'def')
{'x': 'abc', 'y': 'def'}
```

You can also make p1 a default argument as shown below

In [ ]:
```
def function(*args, p1='default', **kwargs):
  print(p1)
  print(args)
  print(kwargs)

function('abc', 'def', x='abc', y='def')
```

```
default
('abc', 'def')
{'x': 'abc', 'y': 'def'}
```

Note: It is good practice to follow a correct order of arguments: non-default, default, args and then kwargs

Function Specifications

A specification of a function defines a contract between the implementer of a function and those who will be writing programs that use the function. We refer to the users of a function as its clients. This contract can be thought of as containing two parts:

Assumptions

These describe conditions that must be met by clients of the function. Typically, they describe constraints on the actual parameters. Almost always, they specify the acceptable set of types for each parameter, and not infrequently some constraints on the value of one or more parameters. For example, the specification of a function to divide two numbers might require that divisor not be zero.

Guarantees

These describe conditions that must be met by the function, provided it has been called in a way that satisfies the assumptions. For example, the specification of a function to divide two numbers might guarantee that it returns the float answer for division of two numbers

In [ ]:
```
def divide(a, b):
  """Assumes a is an integer and b is a non-zero integer.
  Returns a float value which contains the division of a by b."""
  return(a/b)

print(divide(1,3))
```

```
0.3333333333333333
```

The text between the triple quotation marks is called a docstring in Python. By convention, Python programmers use docstrings to provide specifications of functions. These docstrings can be accessed using the built-in function help.

In [ ]:
```
help(divide)
```

```
Help on function divide in module __main__:

divide(a, b)
    Assumes a is an integer and b is a non-zero integer.
    Returns a float value which contains the division of a by b.
```

Recursion

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

Recursion is a common mathematical and programming concept. It means that a function calls itself. This has the benefit of meaning that you can loop through data to reach a result.

Factorial with Recursion

In [ ]:
```
def fact(n):
  if n < 2:
    return 1
  return n * fact(n-1)

print('Factorial of 5 is', fact(5))
```

```
Factorial of 5 is 120
```

Fibonacci Series with Recursion

In [ ]:
```
def fib(n):
  if n < 2:
    return n
  return (fib(n-1) + fib (n-2))

fib(10)
```

Out[ ]:
```
55
```

Program to check happy number with recursion

In number theory, a happy number is a number which eventually reaches 1 when replaced by the sum of the square of each digit.

In [ ]:
```
def happy(n):
  if n == 1 or n == 4:
    return n
  sum = 0
  for i in str(n):
    sum += int(i) ** 2
  return happy(sum)

n = int(input('Enter a number: '))

if happy(n) == 1:
  print('Happy')
else:
  print('Unhappy')
```

```
Enter a number: 13
Happy
```

Write a Python program to create a sequence where the first four members of the sequence are equal to one, and each successive term of the sequence is equal to the sum of the four previous ones. Find the Nth member of the sequence.

In [ ]:
```
def new_seq(n):
    if n<5:
        return 1
    return new_seq(n-1) + new_seq(n-2) + new_seq(n-3) + new_seq(n-4)
print(new_seq(5))
print(new_seq(6))
print(new_seq(7))
```

```
4
7
13
```

Local vs Global Variables

In [ ]:
```
n = 10

def fun(x):
  x = x + 1 #local variable can be accessed only from inside the function
  global n
  n = x + 7

fun(10)
print(n)
```

```
18
```