Python Functions

S1 MCA

- 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:

```
>>>    def my_function():
                print("Hello from a function")
```

Calling a Function

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

```
>>>
        def my_function():
                print("Hello from a function")

        my_function()
```

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.

```
def my_function(fname):
  print(fname + " Refsnes")

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

- *Arguments* are often shortened to *args* in Python documentations.

- **Parameters or Arguments?**
- The terms *parameter* and *argument* can be used for the same thing: information that are passed into a function.
- From a function's perspective:
- A parameter is the variable listed inside the parentheses in the function definition.
- An argument is the value that is sent to the function when it is called.

- **Number of Arguments**
- 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.

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

my_function("Emil", "Refsnes")
```

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

Example

This function expects 2 arguments, but gets only 1:

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

my_function("Emil")
```

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

- *Arbitrary Arguments* are often shortened to **args* in Python documentations.

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 = "Emil", child2 = "Tobias", child3 = "Linus")
```

- The phrase *Keyword Arguments* are often shortened to *kwargs* in Python documentations.

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")
```

Default Parameter Value

- Default Parameter Value
- The following example shows how to use a default parameter value.
- If we call the function without argument, it uses the default value:

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

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

Passing a List as an Argument

- 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.

```
def my_function(food):
  for x in food:
    print(x)

fruits = ["apple", "banana", "cherry"]

my_function(fruits)
```

Return Values

- 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))
```

The pass Statement

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

Positional-Only Arguments

- You can specify that a function can have ONLY positional arguments, or ONLY keyword arguments.
  To specify that a function can have only positional arguments, add , / after the arguments:

```
def my_function(x, /):
  print(x)

my_function(3)
```

- Without the , / you are actually allowed to use keyword arguments even if the function expects positional arguments:

- But when adding the , / you will get an error if you try to send a keyword argument:
-

```
def my_function(x, /):
  print(x)

my_function(x = 3)
```

Keyword-Only Arguments

- To specify that a function can have only keyword arguments, add *, *before* the arguments:

```
def my_function(*, x):
  print(x)

my_function(x = 3)
```

- Without the *, you are allowed to use positional arguments even if the function expects keyword arguments:

```
def my_function(x):
  print(x)

my_function(3)
```

- But with the *, you will get an error if you try to send a positional argument:
-

```
def my_function(*, x):
  print(x)

my_function(3)
```

Combine Positional-Only and Keyword-Only

- You can combine the two argument types in the same function.
- Any argument *before* the / , are positional-only, and any argument *after* the *, are keyword-only.

```
def my_function(a, b, /, *, c, d):
  print(a + b + c + d)

my_function(5, 6, c = 7, d = 8)
```

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.

```
def tri_recursion(k):
  if(k > 0):
    result = k + tri_recursion(k - 1)
    print(result)
  else:
    result = 0
  return result

print("Recursion Example Results:")
tri_recursion(6)
```

Python Lambda

- 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*

-

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

```
x = lambda a, b, c : a + b + c
print(x(5, 6, 2))
```

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

Why Use Lambda Functions?

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

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

mydoubler = myfunc(2)
mytripler = myfunc(3)

print(mydoubler(11))
print(mytripler(11))
```

mytripler is a variable, but it is not holding a number.
It is holding a **function**.

myfunc(3) returns → lambda a: a * 3

So now mytripler **refers to that lambda function**.

So , mytripler = lambda a: a * 3

When you write: mytripler(11)

- You are calling the function stored in mytripler.
- The value 11 is passed as the argument a.
- Inside the lambda: a * 3 → 11 * 3 = 33.