# Introduction to Programming in Python

Lucerne University of
Applied Sciences and Arts

HOCHSCHULE
LUZERN

# Functions

- Functions are a way to organize code. The basic idea is that if you have a piece of code that is likely to be used more than once, you put it in a function so that it may be reused.
- We have already met some functions, such as `print`, `sum`, etc.

In [ ]:
```
sum([1, 2, 3])
```

- A function takes zero or more inputs, called *arguments*. `sum` above takes (at least) one (try what happens when you try to pass zero arguments)
- Other functions take arbitrary numbers of arguments, like `print`:

In [ ]:

```
print()
print(1)
print(1, 2)
```

- When a user calls the function, it (usually) does something with its arguments, and then needs to make the result of that operation available to the user or the surrounding code.
- There are two ways to make the result available:
    - "side effects", and
    - return arguments
- It is important to understand the difference between the two.

Consider again the print function. It makes the result of its action observable by printing to screen:

In [ ]:
```
print(5)
```

Other side effects might include modifying a file, playing a sound, shutting down the computer, etc.

The sum function is different; it doesn't print anything:

In [ ]:
```
a = sum([1, 2, 3])
```

Instead, it *returns* its result. This means that the result can be assigned to a variable, as above. We can then, of course, print the variable:

In [ ]:
```
print(a)
```

The `print` function, on the other hand, doesn't return anything (or rather, it returns `None`, a special data type that represents nothingness):

In [ ]:
```
b = print()
print(b)
```

## Note

Within Jupyter, the difference between printing and returning a result can be difficult to see, because Jupyter notebooks automatically print the value returned by the last expression in a cell, unless you end the line with a semicolon to suppress the output. Consider the following:

In [ ]:
```
sum([1, 2, 3])
```

In [ ]:
```
sum([1, 2, 3]);
```

In [ ]:
```
print(6)
```

In [ ]:
```
print(6);
```

# Defining Functions

- User-defined functions are declared using the `def` keyword:

In [ ]:
```
def mypower(x, y):  # zero or more arguments, here two
    return x**y
```

In [ ]:
```
b = mypower(2, 3)
print(b)
```

Note how the arguments that the user passed are available as the variable `x` and `y` inside the function.

## Exercise

Write a function area that takes two numbers as input (representing the side lenghts of a rectangle), and returns the area of the rectangle.

In [ ]:

## Several Outputs

- Functions can have more than one output argument:

In [ ]:
```
def plusminus(a, b):
    return a+b, a-b
```

In [ ]:
```
c, d = plusminus(1, 2);
print(c, d)
```

## No outputs

A function without a `return` statement will return `None`, like the `print` function.

In [ ]:
```
def greet(name):
    print("Hello", name)
```

In [ ]:
```
a = greet("Simon")
```

In [ ]:
```
print(a)
```

In other words, the absence of a `return` statement is equivalent to

```
return None
```

## Keyword Arguments

- Instead of *positional arguments*, we can also pass *keyword arguments*:

In [ ]:

```
def mypower(x, y):
    return x**y
print(mypower(3, 2))
print(mypower(y=2, x=3) )
```

## Default arguments

- Functions can specify *default arguments*:

In [ ]:
```
def mypower(x, y=2):  # default arguments have to appear at the end
    return x**y
mypower(3)
```

In [ ]:
```
mypower(3, 3)
```

## Exercise, continued

Write a function `area` that takes two numbers as input (representing the side lengths of a rectangle), and returns the area of the rectangle. If the second input is not provided, then the function should compute the area of a square with side length equal to the first input.

Hint: have the second input default to `None`.

Expected output:

```
area(2, 3) # should return 6
area(2)    # should return 4
```

In [ ]:

# Docstrings

Python allows inline documentation via *docstrings*. This is just a string that appears directly after the function definition and documents what the function does:

In [ ]:
```
def mypower(x, y):
    """Compute x^y."""
    return x**y
```

- It is customary to use a triple quoted string; these can contain newlines.
- The docstring is shown by the help function

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

This explains the difference between a comment and a docstring: the former is for the developer, the latter for the user.

## Exercise, continued

Add a docstring to the `area` function.

In [ ]:

## Variable Scope

- Variables defined in functions are local (not visible in the calling scope):

In [ ]:
```
def f():
    z = 1
f()
```

In [ ]:
```
print(z)
```

- The same is true of the input arguments:

In [ ]:
```
def f(num):
    return num**2
```

In [ ]:
```
num
```

Variables defined outside of functions are `global`: they are visible everywhere:

In [ ]:
```
a = 3
def f():
    print(a)
```

In [ ]:
```
f()
```

That is, unless they area "shadowed" by a local variable:

In [ ]:
```
a = 3
def f():
    a = 2
    print(a)
```

In [ ]:
```
f()
print(a)
```

## The `global` statement

If we do actually want to act upon the global variable, then we need to be explicit about it:

In [ ]:
```
a = 3
def f():
    global a
    a = 2
    print(a)
```

In [ ]:
```
f()
print(a)
```

# Quiz

For each of the following, state what gets printed.

1.

```
def f():
    name = "Alexander"
name = "Simon"
f()
print(name)
```

2.

```
def f():
    global name
    name = "Alexander"
name = "Simon"
f()
print(name)
```

3.

```
def f():
    global name
    name = "Alexander"
name = "Simon"
print(name)
```

4.

```
def f(x):
    x = x + 2
x = 7
f(x)
print(x)
```

5.

```
def f(x):
    x[0] = x[0] + 2
    return x
y = [7]
f(y)
print(y[0])
```

# Advanced material on functions

## Mutating functions

- That last example was a bit of a curveball.
- Turns out that if you pass a mutable argument (like a `list`) into a function, then changes to that variable are visible to the caller (i.e., outside the function):

In [ ]:
```
def f(y):
    y[0] = 2
```

In [ ]:
```
x = [1]
f(x)
print(x)
```

## Splatting and Slurping

- Splatting: passing the elements of a sequence into a function as positional arguments, one by one.

In [ ]:

```
def mypower(x, y):
    return x**y
args = [2, 3]  # a list or a tuple
mypower(*args)  # splat (unpack) args into mypower as positional arguments.
```

- Slurping allows us to create *vararg* functions: functions that can be called with any number of positional and/or keyword arguments.

In [ ]:
```
def myfunc(*myargs):
    for i in range(len(myargs)):
        print("Argument number " + str(i+1) + " was " + str(myargs[i]) + ".")
```

In [ ]:
```
myfunc(3, 5)
```

- I.e., The asterisk means "collect all (remaining) positional arguments into the tuple `myargs`".
- This is essentially how the built-in `print` function works.

In [ ]:
```

```

## Recap / further reading (optional)

- https://www.w3schools.com/python/python_functions.asp
- https://python-course.eu/python-tutorial/functions.php

## Homework

- Beginner exercise 16 from https://holypython.com/
- Exercises 2, 3, 6, 9 (hard), 10, 16 from https://www.w3resource.com/python-exercises/python-functions-exercises.php