Install Jupyter Lab or Jupyter Notebook! Then download the zip file from OLAT and open the .ipynb file!
A PDF is provided this week only as a fallback!

About this tutorial

What you're reading now is a Jupyter notebook. It's an interactive document containing a mix of text and Python program snippets. You can edit and run any cell. You may want to modify the python snippets to try how editing them influences the outcome and to get a feel for what is going on.

Since we're going quite fast, you're not expected to remember everything right away. Please review the tutorial at your own pace!

Programming in Python

A program is essentially a "playbook" instructing the computer what to do in different situations. Here is a simple program that outputs a sentence describing Bob's average grade:

In [168…

```
from statistics import mean

def show_mean(name, grades):
    return f"{name.capitalize()}'s mean grade is {mean(grades)}"

print(show_mean("bob", [4.0, 5.25, 6.0, 4.25, 4.0]))
```

```
Bob's mean grade is 4.7
```

If you've installed Python, you could store this program in a raw text file, for example called `grades.py` and then execute it by running `python grades.py`. Python will execute the program statement by statement and then exit.

Interpreter / Shell

Instead of running a whole program, you can also interact with Python interactively, one statement at a time. When you run `python` without any additional arguments, you are dropped into the **interactive** Python shell.

You can follow along with this tutorial either in your own Jupyter notebook or in the interactive Python shell.

If you have neither Jupyter nor Python working yet, you can try using an online Python shell as a stop-gap for now: https://pythonhow.com/python-shell

The Building Blocks of Programming

Before we get into the weeds of how to program in Python, specifically, let's get our bearings. For the purpose of this entry-level course, it's not wrong to say that programming is essentially made up of three main aspects: syntax, data and behavior.

Syntax

Syntax specifies how you must formulate your program. It specifies rules on what each character means in which context. Different programming languages use different syntax. For example, `1 +`

`1` is syntactically correct, while `1 plus 1` is not.

You will learn the Python syntax step by step as you learn new concepts and techniques.

Data

Data can go into a program; data is stored and modified while a program is running, and data can be written out when a program is finished. Data can come in a variety of shapes and sizes:

- Simple data, such as numbers and characters: `0`, `1`, `0.5`, `True`, `False`, `"H"`
- Basic data structures, such as strings: `"Hello"`, lists: `[1,2,"Hello"]`, maps (dictionaries): `{"Police": 117, "Ambulance": 144, "Firefighters": 118}` or sets: `{"Red", "Green", "Blue"}`
- Complex data types (classes and objects) which can represent specific things: `c = Car("Ford", "Focus", 1994); c.honk()`

Behavior

Behavior determines what the program actually does. As a programmer, you can influence the behavior of the program in any number of ways:

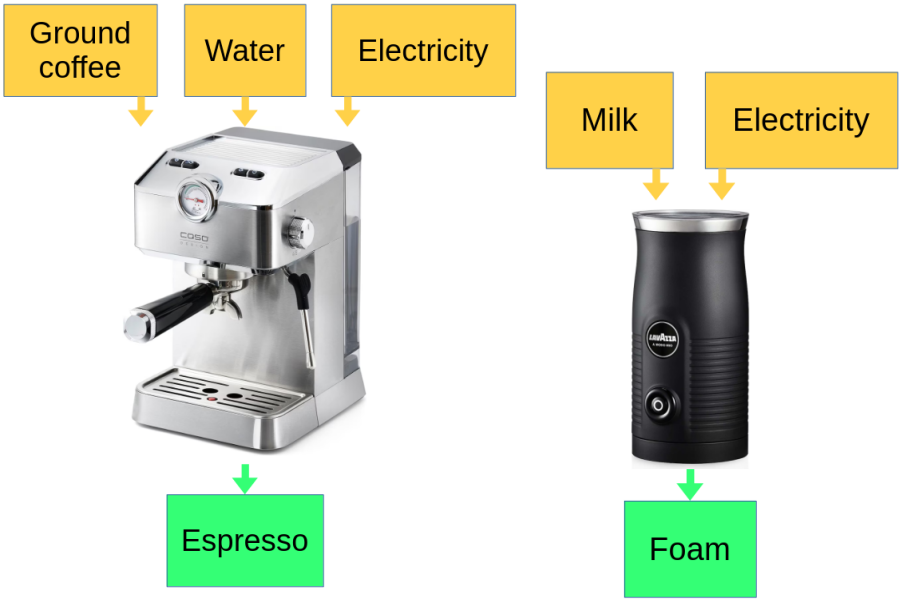
- Functions, like in maths, take parameters as input and produce results a.k.a. "return values" as output.
- Control flow steers behavior based on conditions
- Abstractions & Polymorphism: separating abstract behavior (e.g., "transport") from concrete behavior (e.g., "by train", "on foot", etc.)

In this course, you will learn most of Python's syntax, different ways of representing data, and some techniques to define behavior. However, we will really only scratch the surface of what's possible.

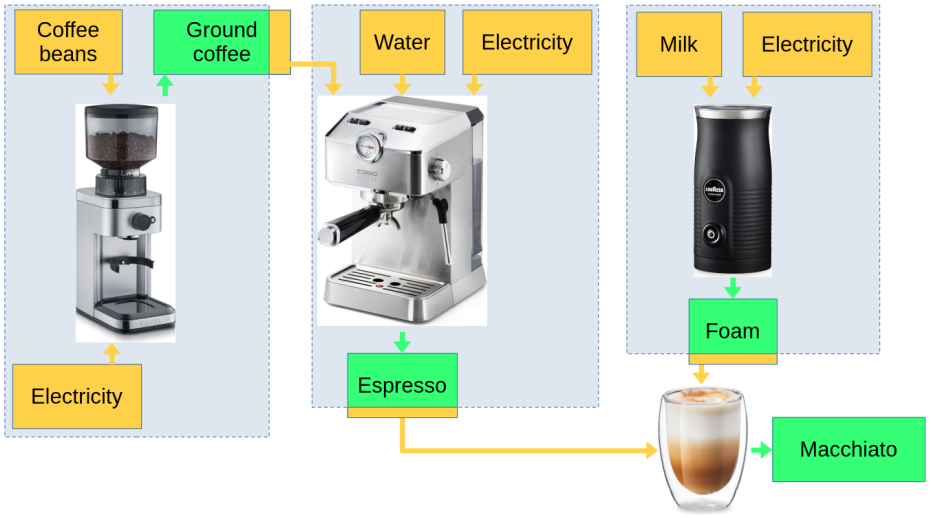
Functions

Functions are the bread and butter of programming in Python, and the most important building block to understand well.

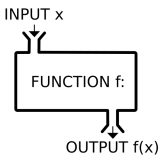
Many real-world processes work based on input and output. For example, input electricity, ground coffee and water into a *coffe maker* function, you'll end up with espresso as output.



Naturally, multiple functions can be chained or otherwise composed:



You already know functions from maths.



Here's how you would write $f(x) = x^2$ in Python:

In [169…
```
def f(x):
    return x**2
```

A function definition starts with `def` followed by the name of the function. A function takes zero or more parameter. In this case, the function takes exactly one parameter, which we call `x`. The function then returns the square of `x`.

Note that just defining this function doesn't really *do* anything. We just defined some functionality, but we haven't used it yet. Let's try using it with a given x now:

In [170…
```
f(6)
```

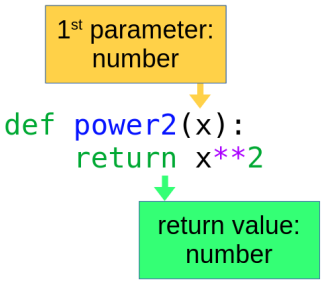
Out[170… 36

Although when programming, you probably want to use a more descriptive name:

In [171…
```
def power2(x):
    return x**2
power2(6)
```

Out[171… 36

From here on out, whenever you see a function, try to visualize or otherwise recognize its features: What it takes and what it returns.



In maths, you know functions mostly as dealing with numbers, receiving some numbers as input, and producing another number as output. However, in the real world, we need to perform all kinds of transformations, not just numerical ones.

Here's a slightly more interesting function that transforms three numbers (year, month and day) into a sentence:

In [172…
```
def birthday_as_string(year, month, day):
    return f"This person was born on {day}.{month}.{year}"
```

It receives these three inputs as parameters, and returns not a number, but a sentence.

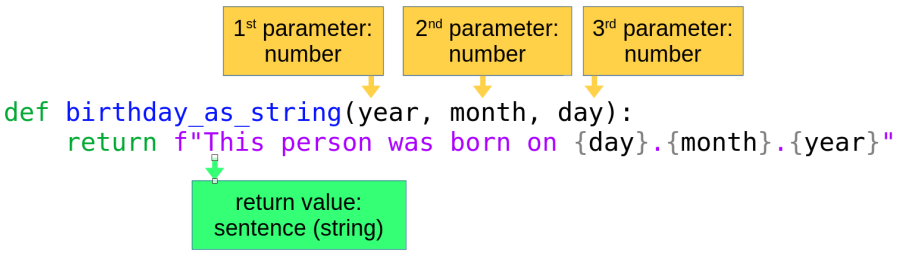
In [173…
```
birthday_as_string(1999,7,13)
```

Out[173… `'This person was born on 13.7.1999'`

In [174…
```
birthday_as_string(2305,7,13)
```

Out[174… `'This person was born on 13.7.2305'`

As you can see, *calling* the `birthday_as_string` function returns a character sequence as expected.



Note that in this particular implementation, Python has **no intrinsic understanding** that it is dealing with a date here! Thus, we could call the function with invalid numbers:

In [175…
```
birthday_as_string(-1999,13,45)
```

Out[175…
```
'This person was born on 45.13.-1999'
```

We decided to name the function `birthday_as_string` and to name the three function parameters `year`, `month` and `day`, but this has no meaning to Python, so the following program behaves exactly the same:

In [176…
```
def bake_cookies(sr71, plums, x):
    return f"This person was born on {sr71}.{plums}.{x}"
bake_cookies(1999,7,13)
```

Out[176…
```
'This person was born on 1999.7.13'
```

Also note that we **must** use the `return` keyword to indicate what the function should return when called. If we forget the `return`, then the function returns *nothing*:

In [177…
```
def birthday_as_string(year, month, day):
    f"This person was born on {day}.{month}.{year}"   # missing return keyword
birthday_as_string(1999,7,13)                         # nothing is printed below b
```

Of course you can have functions that take no parameters. These would be "constant" functions in Maths:

In [178…
```
def pi_to_10_digits():
    return 3.1415926535
pi_to_10_digits()
```

Out[178…
```
3.1415926535
```

Python ships with many built-in functions, which are at your disposal. Here are a few examples:

In [179…
```
min(4,-5,10)              # return the smallest parameter
```

Out[179…
```
-5
```

In [180…
```
round(pi_to_10_digits())  # round to nearest integer number
```

Out[180…
```
3
```

In [181…
```
chr(69)                   # get the unicode character for decimal 69
```

Out[181…
```
'E'
```

In [182…
```
chr(0x1FAE0)              # get the unicode character for hexadecimal 1FAE0
```

Out[182…
```
'🫠'
```

And yet many other functions are hidden away in other *libraries*, for example:

In [183…
```
from statistics import median
median([1, 100, 2, 600, 3])
```

Out[183…
```
3
```

In [184…
```
from os import cpu_count
cpu_count()
```

Out[184…
```
16
```

For now, you need to remember that regular functions in Python:

- Have a name
- Take 0 or more named parameters
- `return` something at the end
- Can be executed, a.k.a. "called", by their name when appending the (zero or more) parameters in braces `(...)`

Exercise

1. Write a function `power` that takes two numbers `base` and `exponent` as parameters and returns the value of `base` to the power of `exponent`. In other words, write $f(x, y) = x^y$ in Python.
2. Call the function to compute 2^8.

In [185…
```
# exercise solution
```

A quick word on imports

You can import functionality from a module called `math` in one of two ways:

You can either import individual members from a module without importing the entire module. For example, if you just want the `cos` function and `pi` from `math`:

In [186…
```
from math import cos, pi

cos(2*pi)
```

Out[186… 1.0

Or you can import the module by name. This allows you to call its functionality using the dot-notation:

In [187…
```
import math

math.cos(2*math.pi)
```

Out[187… 1.0

Also, you could import all the members from a module using the `*` notation. However, note that **this is usually discouraged**. Python follows the mantra of *explicit is better than implicit*, and when importing `*`, you don't explicitely name the members that you wish to use upon importing.

In [188…
```
from math import *

log2(8)
```

Out[188… 3.0

So here, it would have been better to do `from math import log2`, or just `import math` and then use `math.log2`, thus staying *explicit*.

One more thing to note that using imports or dot-notation, you can generally access any member of a module. These members might have different types, for example `math.cos` is a function, but `math.pi` is a floating point number:

In [189…
```
import math
```

```
print(type(math.cos))
print(type(math.pi))
```

```
<class 'builtin_function_or_method'>
<class 'float'>
```

Exercise

There is a module called `random`, which has a member `randrange`, which is a function. This function takes two numbers as parameters, and then returns a random number between those two numbers.

1. Import the `randrange` function from the `random` module
2. Call the `randrange` function with parameters `10` and `21` to generate a random number between 10 and 20

In [190…
```
# exercise solution
```

Expressions

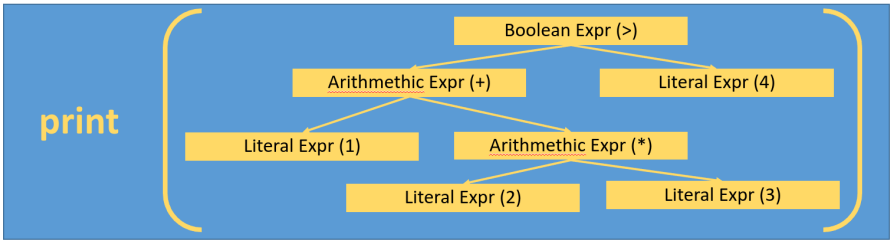
An expression is any contiguous unit that can be executed individually. An expression can consist of several sub-expressions.

Examples for expressions:

- `"Hello"`
- `1`
- `1 + 2` (an expression containing 2 sub-expressions)
- `print(5)` (an expression containing 1 sub-expression)
- `1 + 2 * 3 > 4` (an expression containing 7 sub-expressions)

On the other hand, the following is not an expression, because it cannot stand alone: `1 +`

```
print (1 + 2*3 > 4)
```



If you're interested, you can actually ask Python how it parses any piece of code:

In [191…
```
from ast import dump, parse
print(dump(parse("1 + 2*3 > 4"), indent=4))
```

```
Module(
    body=[
        Expr(
            value=Compare(
                left=BinOp(
                    left=Constant(value=1),
                    op=Add(),
                    right=BinOp(
                        left=Constant(value=2),
                        op=Mult(),
                        right=Constant(value=3))),
                ops=[
                    Gt()],
                comparators=[
                    Constant(value=4)]))],
    type_ignores=[])
```

Types, Values and Variables

Depending on the data type, operations may have a different meaning. Adding two numbers...

In [192…
```
3 + 3
```

Out[192… 6

Is not the same as adding two strings (i.e. text characters):

In [193…
```
"3" + "3"
```

Out[193… '33'

When it comes to working with types and values in python, there are a few things everyone should know:

- The basic scalar types `int`, `float`, (`complex`), `bool` and `NoneType`
- The most commonly used compound types `str`, `tuple`, `list` and `dict`
- The difference between **mutable** and **immutable** values
- How to assign values to variables
- The meaning of **references** (Week 4)

Basic types

In [194…
```
3          # int: integer number
```

Out[194… 3

In [195…
```
3.5        # float: floating point number (not 100% precise!)
```

Out[195… 3.5

In [196…
```
(2 + 3j)   # complex (we're not going to use this)
```

Out[196… (2+3j)

Note that anything after a `#` is considered a comment and ignored by Python.

By the way, you can always figure out the type of a value by using the `type` function:

In [197…
```
type(3.5)
```

Out[197…
```
float
```

A few examples for doing operations on numbers:

In [198…
```
-3       # negation
```

Out[198…
```
-3
```

In [199…
```
1 + 3    # addition
```

Out[199…
```
4
```

In [200…
```
1 - 3    # subtraction
```

Out[200…
```
-2
```

In [201…
```
1 * 3    # multiplication
```

Out[201…
```
3
```

In [202…
```
10 / 3   # division
```

Out[202…
```
3.3333333333333335
```

In [203…
```
10 // 3  # integer division
```

Out[203…
```
3
```

In [204…
```
10 % 3   # modulo
```

Out[204…
```
1
```

In [205…
```
5 ** 3   # exponent
```

Out[205…
```
125
```

Python uses the expected precedence rules:

In [206…
```
1 + 2 * 3
```

Out[206…
```
7
```

Of course, you can use braces to group subexpressions as you wish:

In [207…
```
(1 + 2) * 3
```

Out[207…
```
9
```

Regarding floats, you need to know that computations can be slighly imprecise! We might discuss the reasons later, but for now, be aware that when using floats, the end result may not be perfectly accurate. An example:

In [208…
```
(0.1 + 0.1 + 0.1)
```

Out[208…
```
0.30000000000000004
```

To round floats to integers, you generally want to use the `round` function:

In [209…
```
round(3.5)
```

Out[209…
```
4
```

The `math` module provides additional functions. For example, you could round up (`ceil`) or down (`floor`) explicitely

In [210…
```
import math

math.ceil(1.1)
```

Out[210…
```
2
```

In [211…
```
math.floor(1.9)
```

Out[211…
```
1
```

Note that you can also convert floats to ints using `int` , but this will always **prune** the fractional components, while `floor` always rounds down. This makes a difference when rounding negative numbers:

In [212…
```
int(3.9)
```

Out[212…
```
3
```

In [213…
```
int(-3.9)
```

Out[213…
```
-3
```

In [214…
```
math.floor(-3.9)
```

Out[214…
```
-4
```

A few more examples from the math library:

In [215…
```
print(math.gcd(6,9,81))
print(math.log(math.e))
print(math.log10(100))
print(math.cos(math.pi))
```

```
3
1.0
2.0
-1.0
```

Exercise

Write a function `area` which computes the area of a circle (πr^2) given its radius. The function should take the radius of a circle as the only parameter and it should return the area of the circle as a float. Remember that you can import pi (to great precision) from the math module, rather than defining it yourself.

In [216…
```
# exercise solution
```

Boolean

Programming deals a lot with conditions, which are ultimately either `True` or `False` .

In [217…
```
True
```

Out[217… True

In [218…
```
False
```

Out[218… False

The keywords `and` and `or` are used to create boolean expressions

In [219…
```
True and False
```

Out[219… False

In [220…
```
False or True
```

Out[220… True

And you can use `not` to invert the value:

In [221…
```
not True
```

Out[221… False

Watch out for precedence; `not` binds more strongly than `and` and `or`, but less strongly than `==` and other operators:

In [222…
```
not False and False
```

Out[222… False

In [223…
```
not (False and False)
```

Out[223… True

In [224…
```
not 1 == 2
```

Out[224… True

Expressions separated by `and` and `or` are evaluated left-to right. The evaluation really only goes on as long as necessary. This is called *short-circuiting*. Read up on the documentation: https://docs.python.org/3/library/stdtypes.html#truth-value-testing

Boolean Operations — `and`, `or`, `not`

These are the Boolean operations, ordered by ascending priority:

Operation	Result	Notes
`x or y`	if *x* is true, then *x*, else *y*	(1)
`x and y`	if *x* is false, then *x*, else *y*	(2)
`not x`	if *x* is false, then `True`, else `False`	(3)

Notes:

1. This is a short-circuit operator, so it only evaluates the second argument if the first one is false.
2. This is a short-circuit operator, so it only evaluates the second argument if the first one is true.
3. `not` has a lower priority than non-Boolean operators, so `not a == b` is interpreted as `not (a == b)`, and `a == not b` is a syntax error.

Exercise

Consider the following code snippet, which defines three functions, which all take zero parameters and return a boolean:

In [225…
```
def a():
    print("a")
    return True
def b():
    print("b")
    return False
def c():
    print("c")
    return True
b()
```

```
b
```

Out[225… False

For all of the following expressions, consider:

1. What is the result of the expression?
2. Which of the three functions are actually getting called when executing the expression?

Think about it first, then try executing the expressions. When you run them, you will see which letters are printed on the command line, indicating which of the functions have been called. Make sure you understand why certain functions get called and others do not!

In [226…
```
a() or b() or c()
```

```
a
```

Out[226… True

In [227…
```
(a() or b()) and c()
```

```
a
c
```

Out[227… True

In [228…
```
a() and b() or c()
```

```
a
b
c
```

Out[228… True

In [229…
```
a() or b() and c()
```

```
a
```

Out[229… True

Working with strings

A "string" is a sequence of characters. Python doesn't make a syntactic distinction between strings and individual characters. You can use both double and single quotes for both.

In [230…
```
'Ä'
```

Out[230… 'Ä'

In [231…
```
"Hello"
```

Out[231…
```
'Hello'
```

But note that if you want to use either `'` or `"` as an actual part of the string, you will have to account for that:

In [232…
```
"Hell's kitchen"
```

Out[232…
```
"Hell's kitchen"
```

You could also *escape* the quotation character (by placing a backslash `\` in front of it) if you were to use the same character to start and end the string:

In [233…
```
'This string is surrounded by \' so we have to escape \' inside with a backslash'
```

Out[233…
```
"This string is surrounded by ' so we have to escape ' inside with a backslash"
```

You can see that Jupyter Notebook decides to show this string using double quotes. But there are better ways to handle this than escaping: You can use *triple* single or double quotes, which comes in handy if you're going to use both kinds of quotation marks within the string:

In [234…
```
"""Inside this string, I can use " and ' without any trouble"""
```

Out[234…
```
'Inside this string, I can use " and \' without any trouble'
```

Note that all these variants just use different *syntax*, but are **entirely equivalent** (`==` checks for *equivalence*):

In [235…
```
"Hell's" == 'Hell\'s' == '''Hell's''' == """Hell's"""
```

Out[235…
```
True
```

Python comes with many string manipulation functions. Have a look at the Python documentation to learn more: https://docs.python.org/3/library/stdtypes.html#string-methods

Some examples:

In [236…
```
"hello".upper()
```

Out[236…
```
'HELLO'
```

In [237…
```
"hELLo".lower()
```

Out[237…
```
'hello'
```

In [238…
```
"hello".capitalize()
```

Out[238…
```
'Hello'
```

In [239…
```
"  hello   ".lstrip()
```

Out[239…
```
'hello   '
```

Note an important difference to the functions we mentioned so far (like `birthday_as_string`, `min`, or `round`):

While a function like `round` takes a parameter (`round(3.5)`), the strings functions mentioned here all act directly on the string that we're calling them on: `"hello".upper()`. In other words,

it's not `upper("hello")`. Instead, the `upper` function implicitly receives the calling element (`"hello"`) as its input parameter.

You can also learn all kinds of things about strings using such functions:

In [240…
```
"hello".endswith("lo")
```
Out[240… `True`

In [241…
```
"3".isdigit()
```
Out[241… `True`

To figure out, whether a string contains another string, use the `in` operator:

In [242…
```
"el" in "hello"
```
Out[242… `True`

Exercise

Write a function called `picture_name` which takes a string as the only parameter. The function should return the string with two modifications:

- if the string ends with ".png" (case-insensitive), this should be removed
- the string should be made uppercase

See the given expressions (which should all be `True`) for how the function should behave. Make sure to look at the Python string documentation, since there is a function `removesuffix` which you might want to use here.

In [243…
```
# exercise solution

#print(picture_name("Picture.png") == "PICTURE")
#print(picture_name("Picture.PNG") == "PICTURE")
#print(picture_name("Something else") == "SOMETHING ELSE")
```

Variable assignment

You can assign a value to a variable using the `=` operator:

In [244…
```
name = "Alice"
```

Then you can use this variable in place of an actual value, for example:

In [245…
```
print("Hello " + name + "!")
```
```
Hello Alice!
```

In Python, variables can always be reassigned, so there is no concept of `final`, `val` or `const`, like in some other languages. The previously assigned value is simply lost.

In [246…
```
name = "Alice"
name = "Bob"
print("Hello " + name + "!")
```
```
Hello Bob!
```

You can also assign multiple variables at once using commas, which will come in handy later. Just

make sure that the left and right side contain the same number of variables and values:

In [247…
```
name, age = "Alice", 33
age
```

Out[247… 33

Restrictions on variable and function names

Variable names can be many things, but there are some guidelines and restrictions you must know:

- Typically you want your variable and function names to just contain characters, numbers and `_` (underscore)
- Names are case sensitive (*bananA* is not the same name as *banana*)
- You should use variable names that are meaningful. Excessive use of abbreviations is strongly discouraged.
- Python naming convention for variable and function names is *all-lowercase separated by underscores*
  - Use `birthday_as_string`, not any of `BirthdayAsString`, `birthdayasstring`, or `birthdayAsString`
- Write your code in English.
- You cannot use *reserved keywords* as names:
  - `False`, `await`, `else`, `import`, `pass`, `None`, `break`, `except`, `in`, `raise`, `True`, `class`, `finally`, `is`, `return`, `and`, `continue`, `for`, `lambda`, `try`, `as`, `def`, `from`, `nonlocal`, `while`, `assert`, `del`, `global`, `not`, `with`, `async`, `elif`, `if`, `or`, `yield`
- You should not, and sometimes cannot use certain names in certain contexts, read https://docs.python.org/3/reference/lexical_analysis.html#keywords for more information

String interpolation

Often you will want to print values stored in variables. You've already seen that you can concatenate strings and vairables using `+`:

In [248…
```
print("Hello " + name + "!")
```

```
Hello Alice!
```

However, this quickly becomes tedious and even problematic:

In [249…
```
print(name + " is " + age + " years old")
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[249], line 1
----> 1 print(name + " is " + age + " years old")

TypeError: can only concatenate str (not "int") to str
```

Python complains because it's not possible to add an integer to a string. So instead, we should use one of Python's *three* ways of interpolating strings:

- f-Strings
- "old-style string interpolation" (using `%`)
- the `.format()` function

These days, f-strings are the way to go, so today, we'll only look at those.

In [250…
```
period = "years"
print(f"{name} is {age} {period} old")
```

```
Alice is 33 years old
```

String interpolation also gives you the ability to format individual values differently via the *Format String Syntax* (https://docs.python.org/3/library/string.html#format-string-syntax)

Probably the most common use is to format numbers. Here's an example where a floating point number is formatted, so that the number will be 6 digits long (including the decimal point, padding with leading zeroes if necessary) and have 2 digits after the decimal point:

In [251…
```
age = 33.711922
print(f"{name} is {age:06.2f} years old")
```

```
Alice is 033.71 years old
```

Let's revisit our `birthday_as_string` function:

In [252…
```
def birthday_as_string(year, month, day):
    return f"{day}.{month}.{year}"
birthday_as_string(1952,8,19)
```

Out[252… `'19.8.1952'`

At least around these parts of the world, you would expect the month to always comprise two digits (i.e., with a leading `0`). Using *printf-style* formatting, we can specify that the month should always be formatted using 2 digits, using `0` as the padding character:

In [253…
```
def birthday_as_string(year, month, day):
    return f"{day}.{month:02}.{year}"
birthday_as_string(1952,8,19)
```

Out[253… `'19.08.1952'`

You don't need to know the string formatting syntax by heart! For now, just remember that it exists, and that you don't need to write a lot of code to change how numbers are formatted.

Understanding error messages and finding help

Programming comes with a 100% guarantee for errors and ensuing frustration. Let's review the error message from before:

In [254…
```
name = "Alice"
age = 44
print(name + " is " + age + " years old")
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[254], line 3
      1 name = "Alice"
      2 age = 44
----> 3 print(name + " is " + age + " years old")

TypeError: can only concatenate str (not "int") to str
```

Generally, Python will try to help you out by giving you a code location and error message (typically

the last line). You will want to enter that error into a search engine if you don't know what's happening.

Collections

Collections are composite data structures that can contain other kinds of data and provide *some* structure and functionality out of the box. Different kinds of collections are suitable for different kinds of purposes.

Tuples

Tuples are *array-like* collections which have a **fixed order** and **fixed length**. In Python, a tuple can contain a mix of arbitrary types; duplicate values are fine, too.

In [255…
```
stuff = ("!", 2, 2, 5.5, "Hello", True, None)
stuff
```

Out[255…
```
('!', 2, 2, 5.5, 'Hello', True, None)
```

Indices and Slicing

Use brackets to access tuple or string elements; note that the first element is at index `0`

In [256…
```
stuff[4] + stuff[0]
```

Out[256…
```
'Hello!'
```

You can also use negative indices, which will be counted from the rear:

In [257…
```
stuff[-3] + stuff[-7]
```

Out[257…
```
'Hello!'
```

Instead of just selecting one element, you can also select entire ranges of values:

In [258…
```
stuff[1:4]
```

Out[258…
```
(2, 2, 5.5)
```

Here's an example of using the built-in `sum` function, which takes an iterable (for example, a tuple) of numbers and sums them:

In [259…
```
sum(stuff[1:4])
```

Out[259…
```
9.5
```

Note that of the two numbers used in defining the slice, the first index is included in the slice, while the second index is excluded, i.e. the range `[1:4]` will contain the second, third and fourth element, but not the fifth.

You can also omit the first or last index (or both!) and Python will just go until the beginning or end:

In [260…
```
stuff[:3]
```

Out[260… ('!', 2, 2)

In [261…
```
stuff[:]
```

Out[261… ('!', 2, 2, 5.5, 'Hello', True, None)

Finally, you can get the length of a tuple or string using the `len` function:

In [262…
```
len(stuff)
```

Out[262… 7

All of the above is true for strings as well:

In [263…
```
sentence = "Hello, World!"
sentence[1]
```

Out[263… 'e'

In [264…
```
sentence[-1]
```

Out[264… '!'

In [265…
```
sentence[3:5]
```

Out[265… 'lo'

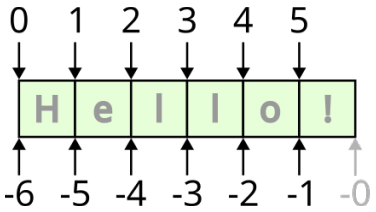
In [266…
```
sentence[-6:]
```

Out[266… 'World!'

In [267…
```
len(sentence)
```

Out[267… 13

Here's how I visualize indices, maybe it also works for you: instead of considering the index numbers to be pointing *at* an element, I consider them pointing at the *gap* just before the referenced element. This works for both positive and negative indices:



Exercise

Write a function `replace_prefix` which takes two strings `sentence` and `prefix` as parameters. The function should replace the first n characters of `sentence` with `prefix` where n corresponds to the length of `prefix`. You may assume that `sentence` is always at least as long as `prefix`. See the given expressions (which should all be `True`) for how the function should behave.

In [268…
```
# exercise solution

#print(replace_prefix("abcdef", "xyz") == "xyzdef")
#print(replace_prefix("What is this?", "This") == "This is this?")
```

Collection unpacking

We saw before that you can assign multiple variables on one line using commas. The same can be done with collections: specify the correct number of variable names on the left side and provide a collection with the same length on the right side.

In [269…
```
record = ("Alice", 33, "alice@example.org")
name, age, email = record
print(email)
```

```
alice@example.org
```

Mutable vs. Immutable

Mutable means that data can be changed; immutable is the opposite: once something has been defined, it cannot be changed. We'll discuss the implications of this in a later lecture, but for now just be aware that some types in Python are inherently immutable.

The most prominent example for an **immutable** type is `str`: Strings cannot actually be changed, once they have been created.

In [270…
```
sentence = "Hello"
print(sentence[0])
```

```
H
```

In [271…
```
sentence[0] = "Y"
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[271], line 1
----> 1 sentence[0] = "Y"

TypeError: 'str' object does not support item assignment
```

But what about the functions like `lower` or `capitalize` that we just saw? These are **not changing** the string itself, they are actually **returning a new** string!

This becomes readily apparent, when we use these functions on a string variable:

In [272…
```
name = "aLiCe"
name.capitalize()
print(name)
```

```
aLiCe
```

It appears, as if `name.capitalize()` did not have any effect! That's because the string stored in `name` cannot be changed, and `capitalize()` does not change it. Strings are immutable! `capitalize()` instead **returns** a new string as a result. To use the changed value, we have to reassign it:

In [273…
```
name = "aLiCe"
name = name.capitalize()
print(name)
```

```
Alice
```

Tuples are also immutable:

In [274…
```
stuff = ("!", 2, 2, 5.5, "Hello", True, None)
print(stuff[1])
```

```
2
```

In [275…
```
stuff[2] = 200
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[275], line 1
----> 1 stuff[2] = 200

TypeError: 'tuple' object does not support item assignment
```

Immutability exists for several good reasons to do with performance and error-avoidance, and we'll discuss them in more detail in a future lecture.

For now, just remember that strings and tuples are immutable.

Lists

Python offers *lists* as an alternative to tuples which *is* mutable. That means that lists have a **fixed order**, but **no fixed length**.

In [276…
```
stuff = ["!", 2, 2, 5.5, "Hello", True, None]
stuff
```

Out[276…
```
['!', 2, 2, 5.5, 'Hello', True, None]
```

Visibly, the only difference is that we use square brackets instead of braces. The usual indexing and slicing operators all work on lists, too:

In [277…
```
print(stuff[4] + stuff[0])
print(stuff[-2:])
```

```
Hello!
[True, None]
```

But in contrast to tuples, this list can be modified. For example, you could re-assign an element at a specific index:

In [278…
```
stuff[0] = "???"
```

Now the first element has been replaced:

In [279…
```
print(stuff)
```

```
['???', 2, 2, 5.5, 'Hello', True, None]
```

You can also add or remove elements:

In [280…
```
stuff.append("More stuff")
stuff
```

Out[280…
```
['???', 2, 2, 5.5, 'Hello', True, None, 'More stuff']
```

In [281…
```
stuff.append(["another list with", "more", "elements"])
stuff
```

Out[281…
```
['???',
 2,
 2,
 5.5,
 'Hello',
 True,
 None,
 'More stuff',
 ['another list with', 'more', 'elements']]
```

In [282…
```
stuff.remove(2)   # Removes the first matching element
stuff
```

Out[282…
```
['???',
 2,
 5.5,
 'Hello',
 True,
 None,
 'More stuff',
 ['another list with', 'more', 'elements']]
```

In [283…
```
del(stuff[2])     # Removes an element at a specific index
stuff
```

Out[283…
```
['???',
 2,
 'Hello',
 True,
 None,
 'More stuff',
 ['another list with', 'more', 'elements']]
```

Tuple or List?

Tuples and lists are very similar. As a rule of thumb, use tuples if you're not going to modify the data. We're going to discuss the intricacies of immutability in another lecture.

Exercise

The following list `source` is given. Use assignment, `add`, `remove` and/or `del`, or any other means to transform the list so that is becomes equivalent to `target`. Do not reassign `source`!

In [284…
```
# exercise solution

source = [2, 1, 1, "Hello", True, print, (1,2,3)]
target = [1, (1,2,3), "Hello", True]

# Modify source until it matches target:
# ...

# You can print source to see the modifications:
print(source)

# This should then be True
source == target
```

```
[2, 1, 1, 'Hello', True, <built-in function print>, (1, 2, 3)]
```

Out[284…
```
False
```

Set

Sets are just that: sets of values in the mathematical sense:

- Values in a set do not have any particular ordering
- Each unique value appears in the set exactly once

In [285…
```
colors = {"Red", "Blue", "Green"}
```

Sets in Python are *mutable*, so you can add and remove elements:

In [286…
```
colors.add("Yellow")
colors.remove("Red")
colors
```

Out[286…
```
{'Blue', 'Green', 'Yellow'}
```

And because it's a set, adding the same value again has no effect:

In [287…
```
colors.add("Yellow")
colors
```

Out[287…
```
{'Blue', 'Green', 'Yellow'}
```

Just like with tuples and lists, you can use the `in` keyword to check if a value is contained in a set:

In [288…
```
"Blue" in colors
```

Out[288…
```
True
```

(Almost) empty collections

If you ever need to create empty collections, you need to know the following syntax. Create an empty list:

In [289…
```
l = []
```

Here's an empty tuple:

In [290…
```
t = ()
```

Tuples are a bit special, because they use the same braces that are used for other things in Python. Consider this:

In [291…
```
t = (3)
```

Is this a tuple containing a single integer? Actually, no:

In [292…
```
t
```

Out[292…
```
3
```

It's just the number `3`. Why? Well, Python cannot really tell the difference between braces used to enclose expressions and braces used to create a tuple.

In [293…
```
t = (((3) + (2)) * (5))
t
```

Out[293…
```
25
```

So if you really want to create a tuple that only contains a single value, you need to use the following syntax:

In [294…
```
t = (3,)
t
```

Out[294…
```
(3,)
```

Similarily, you might notice that an empty dictionary and an empty set would "look" the same, something like `{}` ? So while you can create an empty dictionary using this notation, you will need to use the built-in `set()` function to create an empty set:

In [295…
```
s = set()
s.add("Red")
s
```

Out[295…
```
{'Red'}
```

A note about the shell and `print`

When you type an expression into the Python shell, Python will evaluate the expression and show the resulting value on your command line. This is NOT the same thing as using the `print` function! If you use the print function, it will *also* print to the command line, but the in any case, the *result* of the entered expression is always shown at the end.

The Jupyter notebook works similarily, showing the result of the expression. Understand that only the result of the *entire* code is shown, not each individual expression. In other words, only the result of the last statement is shown. See these examples:

In [296…
```
print("hello")
3.5+1
```

```
hello
```

Out[296…
```
4.5
```

In [297…
```
2+1
5+10
5+4
```

Out[297…
```
9
```

However, this only makes sense in the contex of using Jupyter or the interactive Python shell. If you would save these snippets in a file `example.py` and run it as `python example.py` , then only "hello" would be printed on your command line!

Exercise

1. Evaluate the expression `1+10`
2. Evaluate the expression `'A'`
3. Execute a statement that prints `'A'`

Recognize the reason why both 2. and 3. *look similar*: Python evaluates the expression `'B'` and prints the result, which is just `'B'` . Python also evaluates the expression `print('A')` which explicitely prints the value given to the function (even when not using Jupyter or the interactive Python command line) but otherwise doesn't evaluate to anything, so `A` is all you see.

In [298…
```
# excercise solution
```

What you learned today

- Functions: they take zero or more parameters and return something: **Learn to recognize the input/output (params & types) of a function**
- What is an expression?
- Basic numbers: `int` , `float`
- Booleans, `and` / `or` and *shortcircuiting*
- Check the type of a value using the `type(...)` function
- Imports (`from module import something`)
- How to create and operate on strings
- Variables
- String interpolation (*f-strings*)
- Mutable vs. Immutable
- Collections:
  - Tuples and Lists
  - Sets

If some or all of this is new to you, please take 2-3 hours to go through this tutorial one more time.

1. Modify code snippets to see how the outcome changes
2. Write and execute your own snippets
3. If you run into errors, look up the error message and find the related documentation in the Python API