In [1]:
```
import pybryt
from lecture import pybryt_reference
```

Introduction to Python

Lecture 4

Learning objectives

At the end of this lecture, you will be able to:

- Parse strings to extract specific data of interest.
- Use dictionaries to index data using any type of key.
- Create your own *objects* in Python and develop *member functions* for these new data types.
- Define *special methods* to support operator overloading.
- Explain the difference between *identity* and *equality* of Python objects.
- Make shallow and deep copies of objects.

Python dictionaries

Suppose we need to store the temperatures in Oslo, London, and Paris. The Python list solution might look like:

In [2]:
```
temps = [13, 15.4, 17.5]
# temps[0]: Oslo
# temps[1]: London
# temps[2]: Paris
```

In this case, we need to remember the mapping between the index and the city name. It would be easier to specify the name of the city to get the temperature. Containers such as lists and arrays use a continuous series of integers to index elements. However, for many applications, such an integer index is not useful.

Dictionaries are containers where any immutable Python object (string, hashable tuple, integer, etc.) can be used as an index. Let us rewrite the previous example using a Python dictionary:

In [3]:
```
temps = {"Oslo": 13, "London": 15.4, "Paris": 17.5}
print(f"The temperature in London is {temps['London']}")
```

```
The temperature in London is 15.4
```

Add a new item to a dictionary:

In [4]:
```
temps["Madrid"] = 26.0
print(temps)
```

```
{'Oslo': 13, 'London': 15.4, 'Paris': 17.5, 'Madrid': 26.0}
```

Loop (iterate) over a dictionary:

In [5]:
```
for city in temps:  # please note how we iterate through the keys
    print(f"The temperature in {city} is {temps[city]}")
```

```
The temperature in Oslo is 13
The temperature in London is 15.4
The temperature in Paris is 17.5
The temperature in Madrid is 26.0
```

The index in a dictionary is called the **key**. A dictionary is said to hold key–value pairs (items). We can iterate through those pairs as:

```
for key, value in dictionary.items():
    print(key, value)
```

Does the dictionary have a particular key (i.e. a particular data entry)?

In [6]:
```
if "Berlin" in temps:
    print(f"We have Berlin and its temperature is {temps['Berlin']}")
else:
    print("I don't know Berlin's temperature.")
```

```
I don't know Berlin's temperature.
```

In [7]:
```
print("Oslo" in temps)  # i.e. standard boolean expression
```

```
True
```

The keys and values can be reached as set-like "view" objects:

In [8]:
```
print(f"Keys = {temps.keys()}")
print(f"Values = {temps.values()}")
```

```
Keys = dict_keys(['Oslo', 'London', 'Paris', 'Madrid'])
Values = dict_values([13, 15.4, 17.5, 26.0])
```

In recent versions of Python, dictionaries are guaranteed to return the keys in order they were first inserted into it. If you need them in another order, you will have to run a sort first!

In [9]:
```
for key in sorted(temps):
    value = temps[key]
    print(key, value)
```

```
London 15.4
Madrid 26.0
Oslo 13
Paris 17.5
```

Remove Oslo key-value pair:

In [10]:
```
del temps["Oslo"]  # remove Oslo key w/value
print(temps)
print(len(temps))
```

```
{'London': 15.4, 'Paris': 17.5, 'Madrid': 26.0}
3
```

Similarly to what we saw for arrays, two variables can refer to the same dictionary:

In [11]:
```
t1 = temps
t1["Stockholm"] = 10.0
print(temps)
```

```
{'London': 15.4, 'Paris': 17.5, 'Madrid': 26.0, 'Stockholm': 10.0}
```

So, we can see that while we modified the dictionary bound to `t1`, the dictionary we see from `temps` also changed.

Let us look at a simple example of reading the same data from a file and putting it into a dictionary. We will be reading the file data/deg2.dat.

In [12]:
```
with open("data/deg2.dat", "r") as infile:
    # Start with an empty dictionary
    temps = {}
    for line in infile:
        # If you examine the file you will see a ":" after the city name,
        # so let's use this as the delimiter for splitting the line.
        city, temp = line.split(":")
        temps[city] = float(temp)

print(temps)
```

```
{'Oslo': 21.8, 'London': 18.1, 'Berlin': 19.0, 'Paris': 23.0, 'Rome': 26.0}
```

Similarly to lists, we can also use a **dictionary comprehension** to populate a new dictionary by iterating over a series key:value pairs

In [13]:
```
## Let's convert to Farenheit

farenheit_temps = {city: 1.8*temp+32.0 for city, temp in temps.items()}
print(farenheit_temps)
```

```
{'Oslo': 71.24000000000001, 'London': 64.58000000000001, 'Berlin': 66.2, 'Paris': 73.4, 'Rome': 78.80000000000001}
```

The general syntax of a dictionary comprehension is very similar to lists:

```
somedict = {key: value for element in somelist if test}
```

Exercise 4.1: Make a dictionary from a table

The file data/constants.txt contains a table of the values and the dimensions of some fundamental constants from physics. We want to load this table into a dictionary `constants`, where the keys are the names of the constants. For example, `constants["gravitational constant"]` holds the value of the gravitational constant (6.67259 $\times 10^{-11}$) in Newton's law of gravitation. Make a function `read_constants(file_path)` that reads and interprets the text in the file passed as an argument, and after that returns the dictionary.

In [14]:
```python
# Uncomment and modify the following code. Do not change variable names for testing purposes.
def read_constants(file_path):
    constants = {}
    with open(file_path, "r") as infile:
        next(infile)
        next(infile)

        for line in infile:
            line = line.strip()
            if not line:
                continue
            parts = line.rsplit(maxsplit=2)
            if len(parts) == 3:
                constant, value, dimension = parts
                constants[constant] = float(value)
    return constants

temps = read_constants("data/constants.txt")
print(temps)
```

```
{'speed of light': 299792458.0, 'gravitational constant': 6.67259e-11, 'Planck constant': 6.6260755e-34, 'elementary charge': 1.60217733e-19, 'Avogadro number': 6.0221367e+23, 'Boltzmann constant': 1.380658e-23, 'electron mass': 9.1093897e-31, 'proton mass': 1.6726231e-27}
```

In [15]:
```python
with pybryt.check(pybryt_reference(4, 1)):
    read_constants("./data/constants.txt")
```

```
REFERENCE: exercise-4_1
SATISFIED: False
MESSAGES:
  - SUCCESS: Amazing! You are extracting constants and adding them as items to the dictionary.
  - ERROR: Please think about iterating through the file line by line.
  - SUCCESS: You start with an empty dictionary before the loop. Well done!
  - ERROR: You do not split lines into fragments
  - SUCCESS: Wow! Your resulting dictionary is correct.
```

In [16]:
```
import numbers
import numpy as np

res = read_constants("./data/constants.txt")

assert "Avogadro number" in res.keys()
assert np.isclose(res["speed of light"], 299792458.0)

### BEGIN HIDDEN TESTS
assert isinstance(res, dict)
assert all([isinstance(k, str) and isinstance(v, numbers.Real) for k, v in res.items()])
assert len(res) == 8
### END HIDDEN TESTS
```

Exercise 4.2: Reverse a dictionary

Consider the following dictionary translating some English words to German:

`my_dict = {"dog": "Hund", "cat": "Katze", "house": "Haus", "bicycle": "Fahrrad"}`
Write a Python function `reverse_dict(dictionary)` that takes any dictionary as input and reverses it using **dictionary comprehension**. For instance, if `my_dict` is passed, a German-English dictionary with key-value pairs (items) is returned.

In [17]:
```
# Uncomment and modify the following code. Do not change variable names for testing purposes.
def reverse_dict(dictionary):
    return {value: key for key, value in dictionary.items()}

my_dict = {"dog": "Hund", "cat": "Katze", "house": "Haus", "bicycle": "Fahrrad"}
reversed_dict = reverse_dict(my_dict)
print(reversed_dict)
```

```
{'Hund': 'dog', 'Katze': 'cat', 'Haus': 'house', 'Fahrrad': 'bicycle'}
```

In [18]:
```
with pybryt.check(pybryt_reference(4, 2)):
    reverse_dict({"a": "b", "c": "d", "e": "f", "g": "h"})
```

```
REFERENCE: exercise-4_2
SATISFIED: True
MESSAGES:
  - SUCCESS: Wow! Your function reverses a dictionary correctly.
```

In [19]:
```python
res = reverse_dict({"dog": "Hund", "cat": "Katze", "house": "Haus", "bicycle": "Fahrrad"})

assert "Katze" in res.keys()
assert res["Hund"] == "dog"

### BEGIN HIDDEN TESTS
assert isinstance(res, dict)
assert all([isinstance(k, str) and isinstance(v, str) for k, v in res.items()])
assert len(res) == 4
### END HIDDEN TESTS
```

Exercise 4.3: Compute the area of a triangle

An arbitrary triangle can be described by the coordinates of its three vertices: $(x_1, y_1), (x_2, y_2), (x_3, y_3)$, numbered in a counterclockwise direction. The area of the triangle is given by the formula:

$$A = \frac{1}{2}|x_2y_3 - x_3y_2 - x_1y_3 + x_3y_1 + x_1y_2 - x_2y_1|.$$

Write a function `triangle_area(vertices)` that returns the area of a triangle whose vertices are specified by the argument `vertices`, which is a dictionary and not a list. The keys in the dictionary correspond to the vertex number (1, 2, or 3) while the values are 2-tuples with the x and y coordinates of the vertex - (x, y). For example, for a triangle with vertices $(0, 0)$, $(1, 0)$, and $(0, 2)$ the `vertices` argument is: `{1: (0, 0), 2: (1, 0), 3: (0, 2)}`.

Question: Can the function `triangle_area(vertices)` accept both a nested list and a dictionary as an argument?

In [20]:
```python
# Uncomment and modify the following code. Do not change variable names for testing purposes.

def triangle_area(vertices):
    x1, y1 = vertices[1]
    x2, y2 = vertices[2]
    x3, y3 = vertices[3]
```

```python
    A = 1/2 * abs(x1*(y2 - y3) + x2*(y3 - y1) + x3*(y1 - y2))
    return A

triangle = {1: (0, 0), 2: (1, 0), 3: (0, 2)}
print(triangle_area(triangle))
```

```
1.0
```

In [21]:
```python
with pybryt.check(pybryt_reference(4, 3)):
    triangle_area({1: (100, 20), 2: (101, 130), 3: (-50, 22)})
```

```
REFERENCE: exercise-4_3
SATISFIED: True
MESSAGES:
  - SUCCESS: Wow! Your function computes the triangle area correctly.
```

In [22]:
```python
import numbers
import numpy as np

res = triangle_area({1: (0, 0), 2: (3, 0), 3: (0, 7)})

assert isinstance(res, numbers.Real)
assert np.isclose(res, 10.5)

### BEGIN HIDDEN TESTS
assert triangle_area({1: (0, 0), 2: (0, 0), 3: (0, 0)}) == 0
### END HIDDEN TESTS
```

String manipulation

Text in Python is represented as **strings**. Programming with strings is therefore the key to interpret text in files and construct new text (i.e. **parsing**). First we show some common string operations and then we apply them to real examples. Our sample string used for illustration is:

In [23]:
```python
s = "Berlin: 18.4 C at 4 pm"
```

Strings behave much like tuples (or lists) - they are simply a sequence of characters:

In [24]:
```
print(f"{s[0] = }")
print(f"{s[1] = }")
```

```
s[0] = 'B'
s[1] = 'e'
```

Substrings are just slices of lists and arrays:

In [25]:
```
# from index 8 to the end of the string
print(s[8:])
```

```
18.4 C at 4 pm
```

In [26]:
```
# indices 8, 9, 10 and 11 (not 12!)
# Please remember Python indexing is "first inclusive, last exclusive"
print(s[8:12])
```

```
18.4
```

In [27]:
```
# from index 8 to 8 from the end of the string
print(s[8:-8])
```

```
18.4 C
```

You can also find the start index of a substring:

In [28]:
```
# where does "Berlin" start?
print(s.find("Berlin"))
```

```
0
```

In [29]:
```
print(s.find("pm"))
```

```
20
```

In [30]:
```
print(s.find("Oslo"))
```

```
-1
```

In this last example, `Oslo` does not exist in the list so the return value is -1.

We can also check if a substring is contained in a string:

In [31]:
```
print("Berlin" in s)
```

```
True
```

In [32]:
```
print("Oslo" in s)
```

```
False
```

In [33]:
```
if "C" in s:
    print("C found")
else:
    print("C not found")
```

```
C found
```

Search and replace

Strings also support substituting a substring by another string. In general this looks like `s.replace(s1, s2)`, which gives back a new string replacing occurences of `s1` in `s` by `s2`, e.g.:

In [34]:
```
s = s.replace(" ", "_")
print(s)
```

```
Berlin:_18.4_C_at_4_pm
```

In [35]:
```
s = s.replace("Berlin", "Bonn")
print(s)
```

```
Bonn:_18.4_C_at_4_pm
```

In [36]:
```
# Replace the text before the first colon by "London"
s = s.replace(s[:s.find(":")], "London")
print(s)
```

```
London:_18.4_C_at_4_pm
```

Notice that in all these examples, we assign the new result back to `s`. One of the reasons we are doing this is that strings are constant (i.e immutable) and therefore cannot be modified *inplace*. We **cannot** write for example:

```
s[18] = "5"
TypeError: "str" object does not support item assignment
```

We also encountered examples above where we used the `split()` function to break up a line into separate substrings for a given separator (where a space is the default delimiter). Sometimes we want to split a string into lines - i.e. the delimiter is the carriage return. This can be surprisingly tricky because different computing platforms (e.g. Windows, Linux, MacOS) use different characters to represent a carriage return. For example, Unix uses `\n`. Luckly Python provides a *cross platform* way of doing this so regardless of what platform created the data file, or what platform you are running Python on, it will do the *right thing*:

In [37]:
```
t = "1st line\n2nd line\n3rd line"

print(f"original t =\n{t}")
```

```
original t =
1st line
2nd line
3rd line
```

In [38]:
```
# This works here but will give you problems if you are switching
# files between Windows and either Mac or Linux.
print(t.split("\n"))
```

```
['1st line', '2nd line', '3rd line']
```

In [39]:
```
# Cross platform (i.e. better) solution
print(t.splitlines())
```

```
['1st line', '2nd line', '3rd line']
```

Stripping off leading/trailing whitespace

When processing text from a file and composing new strings, we frequently need to trim leading and trailing whitespaces:

In [40]:
```
s = "        text with leading and trailing spaces          \n"
print(f"-->{s.strip()}<--")
```

```
-->text with leading and trailing spaces<--
```

In [41]:
```
# left strip
print(f"-->{s.lstrip()}<--")
```

```
-->text with leading and trailing spaces
<--
```

In [42]:
```
# right strip
print(f"-->{s.rstrip()}<--")
```

```
-->        text with leading and trailing spaces<--
```

Please note that carriage return is considered as a whitespace character as well.

`join()` - the opposite of `split()`

We can join a list of substrings to form a new string. Similar to `split()`, we put strings together with a delimiter inbetween:

In [43]:
```
strings = ["Newton", "Secant", "Bisection"]
print(", ".join(strings))
```

```
Newton, Secant, Bisection
```

You can prove to yourself that these are inverse operations:

```
t = delimiter.join(stringlist)
stringlist = t.split(delimiter)
```

As an example, let's split off the first two words on a line:

In [44]:
```
line = "This is a line of words separated by space"
words = line.split()
print("words = ", words)
line2 = " ".join(words[2:])
print("line2 = ", line2)
```

```
words =  ['This', 'is', 'a', 'line', 'of', 'words', 'separated', 'by', 'space']
line2 =  a line of words separated by space
```

Exercise 4.4: Improve a program

The file data/densities.dat contains a table of densities of various substances measured in gcm^{-3}. The following program reads the data in this file and produces a dictionary whose keys are the names of substances, and the values are the corresponding densities.

In [45]:
```
def read_densities(filename):
    with open(filename, "r") as infile:
        densities = {}
        for line in infile:
            words = line.split()
            density = float(words[-1])

            if len(words[:-1]) == 2:
                substance = words[0] + " " + words[1]
            else:
                substance = words[0]

            densities[substance] = density

    return densities


densities = read_densities("data/densities.dat")
print(densities)
```

```
{'air': 0.0012, 'gasoline': 0.67, 'ice': 0.9, 'pure water': 1.0, 'seawater': 1.025, 'human body': 1.03, 'limestone': 2.6, 'granite': 2.7, 'iron': 7.8, 'silver': 10.5, 'mercury': 13.6, 'gold': 18.9, 'platinium': 21.4, 'Earth mean': 5.52, 'Earth core': 13.0, 'Moon': 3.3, 'Sun mean': 1.4, 'Sun core': 160.0, 'proton': 280000000000000.0}
```

One problem we face when implementing the program above is that the name of the substance can contain one or two words, and maybe more words in a more comprehensive table. The purpose of this exercise is to use string operations to shorten the code and make it more general. Implement the following two methods in separate functions `read_densities_join` and `read_densities_substrings`, and control that they give the same result.

1. In `read_densities_join`, let *substance* consist of all the words but the last and use the `join()` method to combine the words. Replace any spaces between words in substances with underscore.

2. In `read_densities_substrings`, observe that all the densities (numerical values) start in the same column, and use substrings to divide line into two parts. Replace any spaces between words in substances with underscore. (**Hint**: Remember to strip the first part such that, e.g. the density of ice is obtained as `densities["ice"]` and not `densities["ice     "]`.)

In [46]:
```
# Uncomment and modify the following code. Do not change variable names for testing purposes.
def read_densities_join(filename):
    with open(filename, "r") as infile:
        densities_1 = {}
        for line in infile:
            words = line.split()
            density = float(words[-1])
            substance = "_".join(words[:-1])

            densities_1[substance] = density
    return densities_1

densities1 = read_densities_join("data/densities.dat")
print(densities1)

def read_densities_substrings(filename):
    with open(filename, "r") as infile:
        densities_2 = {}

        for line in infile:
            densities = line[12:].strip()
            substance = line[:12].strip()
            densities_names = substance.replace(" ", "_")
            densities_2[densities_names] = float(densities)

        return densities_2
densities2 = read_densities_join("data/densities.dat")
print(densities2)
```

```
{'air': 0.0012, 'gasoline': 0.67, 'ice': 0.9, 'pure_water': 1.0, 'seawater': 1.025, 'human_body': 1.03, 'limestone': 2.6, 'gran
ite': 2.7, 'iron': 7.8, 'silver': 10.5, 'mercury': 13.6, 'gold': 18.9, 'platinium': 21.4, 'Earth_mean': 5.52, 'Earth_core': 13.
0, 'Moon': 3.3, 'Sun_mean': 1.4, 'Sun_core': 160.0, 'proton': 280000000000000.0}
{'air': 0.0012, 'gasoline': 0.67, 'ice': 0.9, 'pure_water': 1.0, 'seawater': 1.025, 'human_body': 1.03, 'limestone': 2.6, 'gran
ite': 2.7, 'iron': 7.8, 'silver': 10.5, 'mercury': 13.6, 'gold': 18.9, 'platinium': 21.4, 'Earth_mean': 5.52, 'Earth_core': 13.
0, 'Moon': 3.3, 'Sun_mean': 1.4, 'Sun_core': 160.0, 'proton': 280000000000000.0}
```

In [47]:
```
with pybryt.check(pybryt_reference(4, "4_1")):
    read_densities_join("./data/densities.dat")
```

```
REFERENCE: exercise-4_4_1
SATISFIED: True
MESSAGES:
  - SUCCESS: Amazing! You are extracting densities and adding them as items to the dictionary.
  - SUCCESS: Great! You are iterating through the file line by line.
  - SUCCESS: Great! You create an empty dictionary before the loop.
  - SUCCESS: You split lines into fragments. Well done!
  - SUCCESS: Wow! Your implementation of read_densities_join returns the correct dictionary.
```

In [48]:
```
with pybryt.check(pybryt_reference(4, "4_2")):
    read_densities_substrings("./data/densities.dat")
```

```
REFERENCE: exercise-4_4_2
SATISFIED: True
MESSAGES:
  - SUCCESS: Amazing! You are extracting densities and adding them as items to the dictionary.
  - SUCCESS: Great! You are iterating through the file line by line.
  - SUCCESS: Great! You create an empty dictionary before the loop.
  - SUCCESS: You do not split lines into fragments. Well done!
  - SUCCESS: Wow! Your implementation of read_densities_substrings returns the correct dictionary.
```

In [49]:
```
import numbers
import numpy as np

res_join = read_densities_join("./data/densities.dat")
res_substrings = read_densities_substrings("./data/densities.dat")

assert "Earth_core" in res_join.keys()
assert np.isclose(res_join["gold"], 18.9)

assert "Earth_core" in res_substrings.keys()
assert np.isclose(res_substrings["gold"], 18.9)

### BEGIN HIDDEN TESTS
assert isinstance(res_join, dict)
assert all([isinstance(k, str) and isinstance(v, numbers.Real) for k, v in res_join.items()])
assert len(res_join) == 19
```

```python
assert isinstance(res_substrings, dict)
assert all([isinstance(k, str) and isinstance(v, numbers.Real) for k, v in res_substrings.items()])
assert len(res_substrings) == 19
### END HIDDEN TESTS
```

Class: encapsulating variables/data and functions

A class encapsulates variables/data and functions into one single unit. As a programmer, you can create a new class and thereby a new **object type** (similar to those you have already encountered - `int`, `float`, `string`, `list`, `file`, etc.). Once you have created a class you can create many instances of that type as you wish, just as you can have many `int` or `float` objects.

Modern programming makes heavy use of classes and object orientated programming to manage software complexity, making these important concepts to understand. However, for non-trivial applications the design of good abstractions and classes requires careful consideration, otherwise one can unintentionally increase complexity and hurt the performance of your code. Therefore, you should consider this lecture merely as a gentle introduction illustrated with some simple examples.

Representing a function by a class

Consider a function of t with a parameter v_0:

$$ y(t: v_0, g)=v_0t - {1\over2}gt^2 $$

We need both v_0, g and t to evaluate y. How might we implement this?

One option is to assume we will always pass in all variables as arguments:

```python
def y(t, v0, g=9.81):
    return v0*t - 0.5*g*t**2
```

This looks like a reasonable solution when there are only a couple of parameters. But the software complexity quickly gets out of hand as the number of variables increases (I have worked on legacy codes that had function argument lists that were hundreds of lines long because there was no notion of encapsulation!)

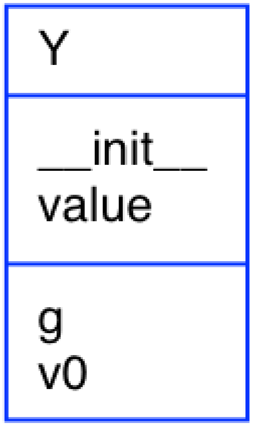
Alternatively we might define `v0` and `g` as global variables:

```
g = 9.81
v0 = ...

...

def y(t):
    return v0*t - 0.5*g*t**2
```

However, the use of global variables is strongly discouraged for many reasons, e.g. very error prone, increased risk of namespace pollution (variables being clobbered when you import a Python module), makes it difficult to manage instances where there might be multiple values for the global variable within the same context, etc.

Let us look at how we might instead implement this as a class.

While we will not cover it in detail here, it is worth noting that professional developers often use UML (Unified Modeling Language) to illustrate the design of a class. Here is a UML diagram for this example:



For this example `class Y` for $y(t: v_0, g)$ has variables `v0` and `g` and a function `value` for computing $y(t: v_0, g)$. Often classes also have the special function `__init__` for initialising class variables.

Here is an implementation of this class:

In [50]:
```
class Y:
    def __init__(self, v0, g=9.81):
        self.v0 = v0
```

```
        self.g = g

    def value(self, t):
        return self.v0*t - 0.5*self.g*t**2
```

An example of its usage:

In [51]:
```
y = Y(v0=3)        # Create instance
v = y.value(0.1)  # Compute function value

print(v)
```

```
0.25095
```

When we write `y = Y(v0=3)` we create a new *instance* of *type* `Y`.

`Y(3)` is a call to the constructor:

```
def __init__(self, v0, g=9.81):
    self.v0 = v0
    self.g = g
```

Think of `self` as `y`, i.e. the new variable to be created. `self.v0` means that we attach a variable `v0` to self (`y`).

```
Y.__init__(y, 3)   # is the Logic behind Y(3)
```

`self` is always the first argument/parameter in a function, but **never** inserted in the call! After `y = Y(3)`, `y` has two variables `v0` and `g`, and we can take a look at these:

In [52]:
```
print(y.v0)
print(y.g)
```

```
3
9.81
```

Functions in classes are called **methods**. Variables in classes are called **attributes**. Therefore, in the above example the `value` *method* was

```
def value(self, t):
    return self.v0*t - 0.5*self.g*t**2
```

Example on a call:

In [53]:
```
v = y.value(t=0.1)
```

`self` is left out in the call (as discussed above), but Python automatically inserts `y` as the `self` argument inside the `value` method. Inside the `value` *method* things *appear* as

```
return y.v0*t - 0.5*y.g*t**2
```

The method `value` has, through `self`, access to the attributes. Attributes are like *global variables* in the class, and any method gets a `self` parameter as its first argument. The method can then access the attributes of the class through `self`.

In summary, `class Y` collects the attributes `v0` and `g` and the method `value` together as a single unit. `value(t)` is function of `t` only, but has access to the class attributes `v0` and `g`.

The great feature of Python is that we can send `y.value` as an ordinary function of `t` to any other function that expects a function `f(t)`:

In [54]:
```
import numpy as np


def table(f, tstop, n):
    """Make a table of t, f(t) values."""
    for t in np.linspace(0, tstop, n):
        print(t, f(t))
```

In [55]:
```
def g(t):
    return np.sin(t)*np.exp(-t)
```

```
table(g, 2*np.pi, 5)  # pass in ordinary function as first argument
```

```
0.0 0.0
1.5707963267948966 0.20787957635076193
3.141592653589793 5.292178668034404e-18
4.71238898038469 -0.008983291021129429
6.283185307179586 -4.573915527954357e-19
```

In [56]:
```
y = Y(6.5)
table(y.value, 2*np.pi, 5)  # pass in class method as first argument
```

```
0.0 0.0
1.5707963267948966 -1.892426272668997
3.141592653589793 -27.990057339009645
4.71238898038469 -78.29289319902196
6.283185307179586 -152.8009338527059
```

Exercise 4.5: Make a class for function evaluation.

Make a class called `F` that implements the function

$$f(x: a, w) = e^{-ax}\sin(wx).$$

A `value(x)` method computes values of f for a given x, while a and w are class attributes as specified as arguments in the class's `__init__` method.

In [57]:
```
# Uncomment and complete this code - keep the names the same for testing purposes.
import numpy as np

class F:
    def __init__(self, a, w):
        self.a = a
        self.w = w

    def value(self, x):
        return np.exp(-(self.a*x)) * np.sin(self.w*x)
```

In [58]:
```
with pybryt.check(pybryt_reference(4, 5)):
    f = F(0.73, 1.14185)
    f.a, f.w, f.value(3)
```

```
REFERENCE: exercise-4_5
SATISFIED: True
MESSAGES:
  - SUCCESS: You initialise the attribute a correctly. Well done!
  - SUCCESS: You initialise the attribute w correctly. Well done!
  - SUCCESS: Wow! Your implementation of the value method is correct.
```

In [59]:
```
import numbers
import numpy as np

f = F(5, 4.1)

assert np.isclose(f.a, 5)
assert np.isclose(f.w, 4.1)
assert np.isclose(f.value(3), -8.052321580865151e-08)

### BEGIN HIDDEN TESTS
assert isinstance(f, F)
assert isinstance(f.a, numbers.Real)
assert isinstance(f.w, numbers.Real)
assert isinstance(f.value(0.1), numbers.Real)
assert callable(f.value)
### END HIDDEN TESTS
```

Exercise 4.6: Make a simple class

Make a class called `Simple` with:

- one attribute, `i`,
- one method `double` that replaces the value of `i` by `2*i`, and
- an `__init__` method that initializes the attribute `i`.

Use the following code snippet to convince yourself that your class is behaving as expected.

```
s1 = Simple(4)
for i in range(4):
    s1.double()
print(s1.i)

s2 = Simple("Hello")
s2.double(); s2.double()
print(s2.i)
s2.i = 100
print(s2.i)
```

In [60]:

```
# Uncomment and complete this code - keep the names the same for testing purposes.
import numpy as np
class Simple:
    def __init__(self, i):
        self.i = i

    def double(self):
        self.i = 2 * self.i

s1 = Simple(4)
for i in range(4):
    s1.double()
print(s1.i)

s2 = Simple("Hello")
s2.double(); s2.double()
print(s2.i)
s2.i = 100
print(s2.i)
```

```
64
HelloHelloHelloHello
100
```

In [61]:

```
with pybryt.check(pybryt_reference(4, 6)):
    simple = Simple(200.51)
    simple.i
    for k in range(10):
```

```
        simple.double()
        simple.i
```

```
REFERENCE: exercise-4_6
SATISFIED: True
MESSAGES:
  - SUCCESS: Your implementation of double method is correct. Well done!
  - SUCCESS: You initialise the attribute a correctly. Well done!
  - SUCCESS: Wow! Your implementation of class Simple is correct.
```

In [62]:
```
import numbers
import numpy as np

s = Simple(2)
for i in range(10):
    s.double()
assert np.isclose(s.i, 2**11)

### BEGIN HIDDEN TESTS
assert isinstance(s, Simple)
assert isinstance(s.i, numbers.Real)
assert s.double() is None  # No return value
assert callable(s.double)
### END HIDDEN TESTS
```

Another class example: a bank account

- **Attributes**:
  - `name` : name of the owner
  - `number` : account number
  - `balance` : balance
- **Methods**:
  - `deposit` : adds amount to `balance`
  - `withdraw` : subtracts amount from `balance`
  - `dump` : pretty pring

In [63]:
```
class Account:
    def __init__(self, name, account_number, initial_amount=0):
        self.name = name
        self.number = account_number
        self.balance = initial_amount

    def deposit(self, amount):
        self.balance += amount  # self.balance += amount is equivalent to self.balance = self.balance + amount

    def withdraw(self, amount):
        self.balance -= amount  # self.balance -= amount is equivalent to self.balance = self.balance - amount

    def dump(self):
        print(f"name: {self.name}, account number: {self.number}, balance: {self.balance}")
```

In [64]:
```
a1 = Account("John Olsson", "19371554951")
a2 = Account("Liz Olsson", "19371564761", 20000)
a1.deposit(1000)
a1.withdraw(4000)
a2.withdraw(10500)
a1.withdraw(3500)
```

In [65]:
```
a1.dump()
```

```
name: John Olsson, account number: 19371554951, balance: -6500
```

In [66]:
```
a2.dump()
```

```
name: Liz Olsson, account number: 19371564761, balance: 9500
```

Exercise 4.7: Extend a class

Add an attribute called `transactions` to the `Account` class given above. The new attribute counts the number of transactions done in the `deposit` and `withdraw` methods. The total number of transactions should be printed in the `dump` method. Write a simple test program to convince yourself transaction gets the right value after some calls to `deposit` and `withdraw`. When an object of class `Account` is created, attribute `transactions` is initialised to 0.

In [67]:
```python
# Uncomment and complete this code - keep the names the same for testing purposes.

class Account:
    def __init__(self, name, account_number, initial_amount=0, transactions=0):
        self.name = name
        self.number = account_number
        self.balance = initial_amount
        self.transactions = transactions

    def deposit(self, amount):
        self.balance += amount
        self.transactions += 1

    def withdraw(self, amount):
        self.balance -= amount
        self.transactions += 1

    def dump(self):
        print(f"name: {self.name}, account number: {self.number}, balance: {self.balance}, transactions: {self.transactions}")
```

In [68]:
```python
with pybryt.check(pybryt_reference(4, 7)):
    account = Account("Marijan", "321321321", initial_amount=2351)
    account.name, account.number, account.balance, account.transactions

    for i in range(5):
        account.deposit(1001)
        account.balance, account.transactions
        account.withdraw(432.3)
        account.balance, account.transactions
```

```
REFERENCE: exercise-4_7
SATISFIED: True
MESSAGES:
  - SUCCESS: Amazing! Your implementation of deposit method is correct.
  - SUCCESS: Amazing! Your implementation of withdraw method is correct.
  - SUCCESS: Wow! You implemented transactions attribute correctly.
  - SUCCESS: You initialise the attribute name correctly. Well done!
  - SUCCESS: You initialise the attribute number correctly. Well done!
  - SUCCESS: You initialise the attribute balance correctly. Well done!
  - SUCCESS: You initialise the attribute transactions correctly. Well done!
```

In [69]:
```python
import numbers
import numpy as np

account = Account("Marijan", "321321321", initial_amount=1000)
assert account.name == "Marijan"
assert account.number == "321321321"
assert account.balance == 1000
assert account.transactions == 0

for i in range(10):
    account.deposit(10)
assert np.isclose(account.balance, 1100)
assert account.transactions == 10

for i in range(10):
    account.withdraw(5)
assert np.isclose(account.balance, 1050)
assert account.transactions == 20

### BEGIN HIDDEN TESTS
assert isinstance(account, Account)
assert isinstance(account.name, str)
assert isinstance(account.number, str)
assert isinstance(account.balance, numbers.Real)
assert isinstance(account.transactions, numbers.Real)
assert account.deposit(2) is None  # No return value
assert account.withdraw(2) is None  # No return value
assert account.dump() is None  # No return value
assert callable(account.deposit)
assert callable(account.withdraw)
assert callable(account.dump)
### END HIDDEN TESTS
```

```
name: Marijan, account number: 321321321, balance: 1050, transactions: 22
```

Protecting attributes

It is not possible in Python to explicitly protect attributes from being overwritten by the calling function, i.e. the following is possible but not intended:

In [70]:
```
a1.name = "Some other name"
a1.balance = 100000
a1.no = "19371564768"
```

Assumptions on correct usage include:

- The attributes should not be modified directly.
- The `balance` attribute can be viewed.
- Changing `balance` is done through with the methods `draw` and `deposit`.

The remedy is to adopt the convention that attributes and methods not intended for use outside the class should be marked as protected by prefixing the name with an underscore (e.g. `_name`). This is just a convention to warn you to stay away from messing with the attribute directly. There is no technical way of stopping attributes and methods from being accessed directly from outside the class.

We rewrite the account class using this convention:

In [71]:
```
class AccountP:
    def __init__(self, name, account_number, initial_amount):
        self._name = name
        self._no = account_number
        self._balance = initial_amount

    def deposit(self, amount):
        self._balance += amount

    def withdraw(self, amount):
        self._balance -= amount

    def get_balance(self):    # NEW - read balance value
        return self._balance

    def dump(self):
        s = f"{self._name}, {self._no}, balance: {self._balance}"
        print(s)
```

In [72]:
```
a1 = AccountP("John Olsson", "19371554951", 20000)
a1.withdraw(4000)
```

In [73]:
```
print(a1._balance)      # it works, but a convention is broken
```

```
16000
```

In [74]:
```
print(a1.get_balance())  # correct way of viewing the balance
```

```
16000
```

In [75]:
```
a1._no = "19371554955"  # if you did this you'd probably lose your job! Don't mess with the convention.
```

Example - a phone book

A phone book is a list of data about persons. Typical data includes: name, mobile phone, office phone, private phone, email. This data about a person can be collected in a class as **attributes**. Think about what kinds of **methods** make sense for this class, e.g.:

- Constructor for initializing name, plus one or more other data
- Add new mobile number
- Add new office number
- Add new private number
- Add new email
- Write out person data

In [76]:
```
class Person:
    def __init__(self, name, mobile_phone=None, office_phone=None, private_phone=None, email=None):
        self.name = name
        self.mobile = mobile_phone
        self.office = office_phone
        self.private = private_phone
        self.email = email

    def add_mobile_phone(self, number):
        self.mobile = number

    def add_office_phone(self, number):
```

```python
        self.office = number

    def add_private_phone(self, number):
        self.private = number

    def add_email(self, address):
        self.email = address

    def dump(self):
        s = self.name + "\n"
        if self.mobile is not None:
            s += f"mobile phone:   {self.mobile}\n"
        if self.office is not None:
            s += f"office phone:   {self.office}\n"
        if self.private is not None:
            s += f"private phone:  {self.private}\n"
        if self.email is not None:
            s += f"email address:  {self.email}\n"
        print(s)
```

In [77]:

```python
p1 = Person("Gerard Gorman", email="g.gorman@imperial.ac.uk")
p1.add_office_phone("49985")

p2 = Person("ICT Service Desk", office_phone="49000")
p2.add_email("service.desk@imperial.ac.uk")

phone_book = {"Gorman": p1, "ICT": p2}
for p in phone_book:
    phone_book[p].dump()
```

```
Gerard Gorman
office phone:   49985
email address:  g.gorman@imperial.ac.uk

ICT Service Desk
office phone:   49000
email address:  service.desk@imperial.ac.uk
```

Example - a circle

A circle is defined by its center point x_0, y_0 and its radius R. These data can be attributes in a class. Possible methods in the class are `area` and `circumference`. The constructor initializes x_0, y_0 and R.

In [78]:
```python
class Circle:
    def __init__(self, R, x0, y0,):
        self.x0, self.y0, self.R = x0, y0, R

    def area(self):
        return np.pi * self.R**2

    def circumference(self):
        return 2*np.pi*self.R
```

In [79]:
```python
c = Circle(2, -1, 5)
print(f"A circle with radius {c.R} at ({c.x0}, {c.y0}) has area {c.area()}")
```

```
A circle with radius 2 at (-1, 5) has area 12.566370614359172
```

Exercise 4.8: Make a class for straight lines

Make a class called `Line` whose constructor takes two points `p0` and `p1` (2-tuples or 2-lists) as input. The line goes through these two points (see function `line` defined below for the relevant formula of the line). A `value(x)` method computes the `y` value on the line at the point `x` or returns `None` if the line is vertical (i.e. if `(x1-x0) == 0`).

```python
def line(x0, y0, x1, y1):
    """
    Compute the coefficients a and b in the mathematical
    expression for a straight line y = a*x + b that goes
    through two points (x0, y0) and (x1, y1).
    x0, y0: a point on the line (floats).
    x1, y1: another point on the line (floats).
    return: coefficients a, b (floats) for the line (y=a*x+b).
    """
    try:
        a = (y1 - y0)/(x1 - x0)
        b = y0 - a*x0
    except ZeroDivisionError:
```

```
            a, b = None, None

        return a, b
```

In [80]:
```
# Uncomment and complete this code - keep the names the same for testing purposes.
class Line:
    def __init__(self, p0, p1):
        self.p0 = p0
        self.p1 = p1

        x0, y0 = p0
        x1, y1 = p1

        try:
            self.a = (y1 - y0) / (x1 - x0)
            self.b = y0 - self.a * x0
        except ZeroDivisionError:
            self.a = None
            self.b = None

    def value(self, x):
        if self.a is None:
            return None
        return self.a * x + self.b
```

In [81]:
```
with pybryt.check(pybryt_reference(4, 8)):
    line = Line(p0=(123.1, 251.6), p1=(44.3, 12.9))
    line.p0, line.p1, line.value(3.141)
```

```
REFERENCE: exercise-4_8
SATISFIED: True
MESSAGES:
  - SUCCESS: You initialise the attribute p0 correctly. Well done!
  - SUCCESS: You initialise the attribute p1 correctly. Well done!
  - SUCCESS: Amazing! You compute the coefficient a correctly.
  - SUCCESS: Amazing! You compute the coefficient b correctly.
  - SUCCESS: Wow! Your implementation of method value correct.
```

In [82]:
```
import numbers
import numpy as np
```

```
# Undefined line
line = Line(p0=(0, 0), p1=(0, 0))
assert line.p0 == (0, 0)
assert line.p1 == (0, 0)
assert line.value(10) is None

# Well-defined line
line = Line(p0=(0, 0), p1=(10, 10))
assert line.p0 == (0, 0)
assert line.p1 == (10, 10)

for x in np.linspace(-10, 10, 20):
    assert np.isclose(line.value(x), x)

### BEGIN HIDDEN TESTS
assert isinstance(line, Line)
assert isinstance(line.p0, tuple)
assert isinstance(line.p1, tuple)
assert all([isinstance(i, numbers.Real) for i in line.p0])
assert all([isinstance(i, numbers.Real) for i in line.p1])
assert isinstance(line.value(100), numbers.Real)
assert callable(line.value)
### END HIDDEN TESTS
```

Exercise 4.9: Make a class for quadratic functions

Consider a quadratic function $f(x; a, b, c) = ax^2 + bx + c$. Make a class called `Quadratic` for representing f, where a, b, and c are attributes, and the methods are:

1. `value(self, x)` for computing a value of f at a point x,
2. `table(self, L, R, n)` for writing out a table of x and f values for n values of x in the interval $[L, R]$,
3. `roots(self)` for computing the two roots and returning them both in a tuple `(x1, x2)`.

In [83]:
```
# Uncomment and complete this code - keep the names the same for testing purposes.
import numpy as np
class Quadratic:
```

```python
    def __init__(self, a, b, c):
        self.a, self.b, self.c = a, b, c

    def value(self, x):
        return self.a*(x**2) + self.b*x + self.c

    def table(self, L, R, n):
        N = np.linspace(L, R, n)
        for i in N:
            print('x: %g\tf: %g' % (x, self.value(x)))

    def roots(self):
        D = self.b**2 - 4*self.a*self.c
        if D < 0:
            return None
        sqrtD = np.sqrt(D)
        x1 = (-self.b + sqrtD) / (2*self.a)
        x2 = (-self.b - sqrtD) / (2*self.a)
        return (x1, x2)
```

In [84]:

```python
with pybryt.check(pybryt_reference(4, 9)):
    f = Quadratic(a=10.2, b=5.6, c=-30.11)
    f.a, f.b, f.c, f.value(500), f.roots()
```

```
REFERENCE: exercise-4_9
SATISFIED: True
MESSAGES:
  - SUCCESS: You initialise the attribute a correctly. Well done!
  - SUCCESS: You initialise the attribute b correctly. Well done!
  - SUCCESS: Great! You implemented value method correctly.
  - SUCCESS: Your computed root x1 is correct. Amazing!
  - SUCCESS: Your computed root x2 is correct. Amazing!
  - SUCCESS: Wow! Your implementation of class Quadratic is correct.
```

In [85]:

```python
import numbers
import numpy as np

f = Quadratic(a=5, b=6, c=1)
assert f.a == 5
assert f.b == 6
assert f.c == 1
```

```
assert f.value(0) == 1
assert np.allclose(f.roots(), (-0.2, -1))
assert f.table(0, 10, 11) is None

### BEGIN HIDDEN TESTS
assert isinstance(f, Quadratic)
assert isinstance(f.a, numbers.Real)
assert isinstance(f.b, numbers.Real)
assert isinstance(f.c, numbers.Real)
assert isinstance(f.value(100), numbers.Real)
assert all([isinstance(i, numbers.Real) for i in f.roots()])
assert callable(f.value)
assert callable(f.table)
assert callable(f.roots)
### END HIDDEN TESTS
```

```
x: 10   f: 561
x: 10   f: 561
x: 10   f: 561
x: 10   f: 561
x: 10   f: 561
x: 10   f: 561
x: 10   f: 561
x: 10   f: 561
x: 10   f: 561
x: 10   f: 561
x: 10   f: 561
```

Special methods

Some class methods have leading and trailing double underscores. You have already met one of these, `__init__` used to initialise an object upon creation. Other examples include `__call__(self, ...)` and `__add__(self, other)`. These *special methods* enable more elegant abstractions and interfaces. Consider for example the difference between the equivalent statements:

```
y = Y(4)
```

rather than

```
y = Y
Y.__init__(Y, 4)
```

Special member function, `__call__`: make the class instance behave and look as a function

Let us replace the `value` method in `class Y` by a `__call__` special method:

In [86]:
```
class Y:
    def __init__(self, v0, g=9.81):
        self.v0 = v0
        self.g = g

    def __call__(self, t):
        return self.v0*t - 0.5*self.g*t**2
```

Now we can write:

In [87]:
```
y = Y(3)
v = y(0.1)  # same as v = y.__call__(0.1)
```

The instance y behaves/looks as a function! The `value(t)` method in the first example does the same, but the special method `__call__` provides a more elegant and concise syntax for computing function values.

Special member function, `__str__`: represent object as a string for printing

In Python, we can usually print an object `a` by `print(a)`. This works for built-in types (strings, lists, floats, ...). However, if we have made a new type through a class, Python does not know how to print objects of this type. However, if the class has defined a method `__str__`, Python will use this method to convert the object to a string.

In [88]:
```
class Y:
    def __init__(self, v0, g=9.81):
        self.v0 = v0
        self.g = g

    def __call__(self, t):
        return self.v0*t - 0.5*self.g*t**2
```

```
    def __str__(self):
        return f"{self.v0}*t - 0.5*{self.g}*t**2"
```

In [89]:
```
y = Y(1.5)
print(y)
```

```
1.5*t - 0.5*9.81*t**2
```

Special methods for overloading arithmetic operations

```
c=a+b              # c = a.__add__(b)
c=a-b              # c = a.__sub__(b)
c = a*b            # c = a.__mul__(b)
c = a/b            # c = a.__div__(b)
c = a**e           # c = a.__pow__(e)
```

Special methods for overloading conditional operations

```
a == b              #  a.__eq__(b)
a != b              #  a.__ne__(b)
a < b               #  a.__lt__(b)
a <= b              #  a.__le__(b)
a > b               #  a.__gt__(b)
a >= b              #  a.__ge__(b)
```

Equality vs. identity

Before we discuss what the difference between *equality* and *identity* in Python is, let us first think about what happens in the background when we write `a = [1, 2, 3]`.

In [90]:
```
a = [1, 2, 3]
```

On the left hand side, we have a variable name `a` and on the right hand side, we have a list literal. Python first creates a list object `[1, 2, 3]` in memory. Then, it *binds* the variable `a` to it. Therefore, the object we created in memory is accessible to us via variable `a` - we also say that `a` is a reference to the object.

Each object created in memory has a unique ID. The ID of our newly created list is:

In [91]:
```
print(f"{id(a) = }")
```

```
id(a) = 128515037417408
```

If you have experience in other programming languages (e.g. C/C++) you might be used to the idea that `a = 2` places the value `2` to a "box" in memory named `a`. Similarly, in C/C++, if we write `b = a`, we copy the value from box `a` to box `b`. We end up with having `2` stored in two memory locations. Therefore, changing the value of `b` should not affect the value stored in `a` since they are two distinct memory locations. Let us see if that is the case in Python:

In [92]:
```
a = [1, 2, 3]  # we create the list and bind variable "a" to it
b = a
b.append(4)

print(f"{a = }")
print(f"{b = }")
```

```
a = [1, 2, 3, 4]
b = [1, 2, 3, 4]
```

`b = a` did not make any copies - it bound variable `b` to the same object `a` was referring to. Therefore, instead of having two lists in memory - one at location `a` and another at location `b`, we have one object in memory we can access via both `a` and `b`.

Let us compare their IDs. If they are the same, then both `a` and `b` refer to the same object.

In [93]:
```
print(id(a) == id(b))  # both a and b are referring to the same object
```

```
True
```

In Python, we compare whether two variables point to the same obejct using `is`. `id(a) == id(b)` is equivalent to `a is b`:

In [94]:
```
print(a is b)
```

```
True
```

Alternatively we could check the opposite:

In [95]:
```
print(a is not b)
```

```
False
```

If `a is b` results in `True`, that means that both `a` and `b` are bound to the same object. If we change the object via `a`, then, when we attempt to access it via `b` we get the changed object.

Let us have a look at a different example:

In [96]:
```
c = [1, 2, 3]
d = [1, 2, 3]
```

Are variables `c` and `d` pointing to the same object? How can we check that? We compare their IDs:

In [97]:
```
print(f"{id(c) = }")
print(f"{id(d) = }")
print(f"{c is d = }")
```

```
id(c) = 128515037123072
id(d) = 128515037425280
c is d = False
```

They are not bound to the same object. Although two equal lists are created in memory, they are actually two different objects in memory (stored at two different locations). They are *equivalent* but not *identical*.

As we have seen previously, we check the identity usinf `is`, but the equivalence using binary `==` relational operator"

In [98]:
```
print(c == d)
```

```
True
```

The equivalence in Python is defined by the special `__eq__` method in our class definition. If `__eq__` is not defined, Python would then perform the identity check by comparing object IDs.

Since `c` and `d` are not two equivalent objects, changing the value of `c` should not change the value `d`:

In [99]:
```
c.append(4)  # we modify c

print(f"{c = }")
print(f"{d = }")
```

```
c = [1, 2, 3, 4]
d = [1, 2, 3]
```

As always in life, there are special cases. Let us have a look at one. Are the IDs of variables `e` and `f` defined in the next cell different or the same? What do you think?

In [100…
```
e = 0
f = 0
```

We now know how to check that: using `is`.

In [101…
```
print(e is f)
```

```
True
```

Based on the previous examples, we would expect `e` and `f` to be bound to different objects. However, they are not. This is because Python creates some objects in advance in memory for optimisation to avoid having thousands of copies of the same object in memory. This is called *interning*).

One such example is `None`, which is why you might be warned by linters to write

```
if a is None:  # or a is not None
    pass
```

instead of

```
if a == None:
    pass
```

Discussion of interning and internal implementation of Python interpreter is not the topic of this lecture, but having some understanding of what happens in the background when we write `a = 5` or `a = b` could be very helpful when debugging your code.

Are tuples really immutable?

In lecture 2, we have shown that tuples are not actually immutable and promised we will discuss this further in this lecture. We looked at the following example:

In [102…
```
my_tuple = (1, 2, [3, 4])
my_tuple[2].append(5)
print(f"{my_tuple = }")
```

```
my_tuple = (1, 2, [3, 4, 5])
```

A tuple is immutable, but we modified it. How is that possible?

Immutability means we cannot change the ID of any of tuple's elements. However, we can change the object itself:

In [103…
```
my_tuple = (1, 2, [3, 4])
print(f"Before: {id(my_tuple[2]) = }")  # ID of an object the second element is pointing to

my_tuple[2].append(5)  # we change the list
print(f"After:  {id(my_tuple[2]) = }")  # has the ID changed?
```

```
Before: id(my_tuple[2]) = 128515037423232
After:  id(my_tuple[2]) = 128515037423232
```

We can see that the ID of the list object in memory has not changed. This is in accordance with tuple immutability rules. Although object's ID has not changed, the object itself (`my_tuple[2]`) has. Because of that, we say that the tuple is immutable if and only if all its elements are immutable. The best way for us to check if a tuple is immutable is by computing its hash:

```
>>> hash(my_tuple)

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [137], in <cell line: 1>()
----> 1 hash(my_tuple)

TypeError: unhashable type: 'list'
```

The error we get tells us that our tuple is not hashable since it contains a list that can be changed as we have seen before. On the other hand, if a tuple contains only immutable elements:

In [104…
```
my_tuple2 = (1, 2, "a", (3, 4))

print(f"{hash(my_tuple2) = }")
```

```
hash(my_tuple2) = -6357896215600669351
```

Passing variables to functions

Let us have a look at how variables are passed to functions:

In [105…
```
a = [1, 2, 3]  # mutable type


def add_element(sequence, element):
    sequence.append(element)

    return None  # the function does not return anything to the caller's scope
```

In [106…
```
print(f"Before call: {a = }")
add_element(a, 4)
print(f"After call: {a = }")
```

```
Before call: a = [1, 2, 3]
After call: a = [1, 2, 3, 4]
```

By passing list `a` via variable `sequence` to function `add_element`, we did not make a copy of the value `a` to `sequence`. We only bound another variable to the same object. Therefore, although variable name `sequence` is in function's local scope, we can change the "global" object it is referring to.

This might be confusing to C/C++ programmers since they would expect the variable to be passed by value. In other words, a copy of the variable would be made to the function's local scope. Unless a local value is returned to the caller's scope, there is no way to know whether the function changed the passed value or not. In Python, for C/C++ programmers, this would look similar to *passing by reference*.

Copying objects

Sometimes, we do not want our variable to be bound to the same object. Instead, we want to make a copy of the object in memory and have two equivalent but not identical objects. There are several ways to do so in Python. One way is to use `copy` - a module from Python's standard library. For instance:

In [107…
```
a = [1, 2, 3]
b = a

print(f"Are a and b identical (a is b): {a is b}")
print(f"Are a and b equal (a == b): {a == b}")
```

```
Are a and b identical (a is b): True
Are a and b equal (a == b): True
```

Let us now use `copy.copy`:

In [108…
```
import copy

a = [1, 2, 3]
b = copy.copy(a)

print(f"Are a and b identical (a is b): {a is b}")
print(f"Are a and b equal (a == b): {a == b}")
```

```
Are a and b identical (a is b): False
Are a and b equal (a == b): True
```

By making a copy using `copy.copy` method, we created another object in memory that is equivalent to the original one. Now, by changing `b`, we would not be changing the object `a` is referring to:

In [109…
```
b.append(4)

print(f"{a = }")
print(f"{b = }")
```

```
a = [1, 2, 3]
b = [1, 2, 3, 4]
```

Shallow vs. deep copy

When we call `b = copy.copy(a)` on an object, does that mean we can never change `a` by making changes in `b`? Let us have a look:

In [110…
```
sublist = [1, 2]
list_a = [sublist, 3, 4]

print(f"{list_a = }")
```

```
list_a = [[1, 2], 3, 4]
```

Now, we can make a copy of that list:

In [111…
```
list_b = copy.copy(list_a)
```

We expect the IDs of `list_a` and `list_b` to be different - they are two different objects in memory:

In [112…
```
print(list_a is list_b)
```

```
False
```

Let us now make a change in `list_b` and see whether that made a change in `list_a`:

In [113…
```
list_b[0].append('HUH?')

print(f"{list_a = }")
print(f"{list_b = }")
```

```
list_a = [[1, 2, 'HUH?'], 3, 4]
list_b = [[1, 2, 'HUH?'], 3, 4]
```

How is that possible? When we made a copy, we created a new list in memory and copied the references of elements. All copied references are bound to the same object:

In [114…
```python
print(list_a[0] is list_b[0])
```

```
True
```

We say we did a shallow copy by calling `copy.copy`. Alternatively, we could have called `copy.deepcopy` to make copies of objects elements are bound to as well.

In [115…
```python
sublist = [1, 2]
list_a = [sublist, 3, 4]
list_b = copy.deepcopy(list_a)

list_b[0].append('HUH?')

print(f"{list_a = }")
print(f"{list_b = }")
```

```
list_a = [[1, 2], 3, 4]
list_b = [[1, 2, 'HUH?'], 3, 4]
```

By making a deep copy, we ensured that all element objects are copied:

In [116…
```python
print(list_a[0] is list_b[0])
```

```
False
```

Please note that there is no guarantee a copy would be made as you expect. When `copy.copy(my_object)` is called, a special method `__copy__` is called from `my_object`. Therefore, the behaviour of copy depends on the implementation of `__copy__` special method.

Most of the time, this is something you, a Python programmer, do not have to worry about. However, having some knowledge of basic Python concepts such as identity, equality, copying, etc. can be very helpful when debugging your code.

In [ ]: