Introduction to Data Science for Social Scientists

Lesson 3

New Data Types: NumPy arrays + Intro to NumPy

In [ ]:
```
## Let's say we have two lists that hold information on how much money we spent...
## on groceries and entertainment during each day of the month - in RS Dinars

expenses_groceries = [3500, 0, 0, 1200]
expenses_entertainment = [1000, 0, 0, 500]

## It would be really nice to get an overview of how much money in total we spent...
## ... per day.

## Simple, right?

expenses_groceries + expenses_entertainment
```

Out[ ]:
```
[3500, 0, 0, 1200, 1000, 0, 0, 500]
```

Nope, that doesn't work. At least, not the way we wanted.

Remember string concatenation?

Remember how different functions and operators work differently for different data types?

"+" operator, when used with lists, results in list concatenation.

If we want a pairwise summation of elements in two lists, we need the help of a package called NumPy.

But first, what are packages?

- Packages are collections of objects, such as data types and functions that aren't available in base Python, but can quickly be imported to enhance its abilities. We use them all the time. They make life much, much easier!
- Why don't they exist in base Python?

  What is NumPy and why it's important?

In [ ]:
```
import numpy as np ## This is how we import a package.
## "np" is numpy's commonly used alias

my_list = [1, 2, 3, 4]
np.array(my_list)
```

Out[ ]:
```
array([1, 2, 3, 4])
```

In [ ]:
```
my_array = np.array(my_list)
type(my_array)
```

Out[ ]:
```
numpy.ndarray
```

In [ ]:
```
list_groceries = [3500, 0, 0, 1200]
list_entertainment = [1000, 0, 0, 500]

array_grocesies = np.array(expenses_groceries)
array_enterntainment = np.array(expenses_entertainment)

arr_sum = array_grocesies + array_enterntainment
arr_sum ## It works like a charm this time!
```

Out[ ]:
```
array([4500,    0,    0, 1700])
```

In [ ]:
```
## list_groceries = [3500, 0, 0, 1200] ## Just to remember
## list_entertainment = [1000, 0, 0, 500] ## Just to remember

## Okay, now we know how pairwise summation works.
## What about substraction?

print(array_grocesies - array_enterntainment)

## Multiplication?
print(array_grocesies * array_enterntainment)
```

```
[2500    0    0  700]
[3500000       0       0  600000]
```

In [ ]:
```
## Division?
print(array_grocesies / array_enterntainment)
```

```
[3.5 nan nan 2.4]

c:\Users\PC\anaconda3\lib\site-packages\ipykernel_launcher.py:2: RuntimeWarning: invali
d value encountered in true_divide
```

Python lists VS NumPy arrays - similarities and differences

🏠 Algebra: vector and a scalar

🏠 Algebra: vectors of unequal lengths

🏠 Indexing and slicing arrays

🏠 Arrays with more than 1 dimension

New Data Types: Dictionary

In [ ]:
```
## In real life, we often deal with more complex data structures
## How can we express them in Python?
## 1) With lists?

list_first_names = ["Ana", "Petar", "Marko", "Jovana"]
list_professions = ["Accountant", "Programmer", "Manager", "Junior Developer"]

## Doesn't look so good - not very easy to work with

list_professions[list_first_names.index("Ana")]
```

Out[ ]:
```
'Accountant'
```

In [ ]:
```
## 2) With lists of lists?

list_employees = [["Ana", "Accountant"],
                  ["Petar", "Programmer"],
                  ["Marko", "Manager"],
                  ["Jovana", "Junior Developer"]]

## Still not so good...

for sublist in list_employees:
    if "Ana" in sublist:
        print(sublist[sublist.index("Ana") + 1])
```

```
Accountant
```

In [ ]:
```
## 3) Say hello to dictionaries!

dict_employees = {"Ana" : "Acountant",
                  "Petar" : "Programmer",
                  "Marko" : "Manager",
                  "Jovana" : "Junior Developer"}

dict_employees["Ana"]
```

Out[ ]:
```
'Acountant'
```

In [ ]:
```
type(dict_employees)
```

Out[ ]:
```
dict
```

Dictionaries are defined with `{key : value}` notation

(unlike lists: `[element1, element2, …]`).

They are a bit like named lists.

Each key in a dictionary corresponds to exactly one value.

Dictionaries can contain different data types.

In [ ]:
```
## Dictionary with different datatypes 1 -> strings, integers, lists, ...

my_dict = {"course name" : "Intro to Python for Social Scientists",
           "starting_date" : "2023.04.02",
           "num_participants" : 20,
           "participant_names" : ["Marko", "Ana", "Petar", "Jovana"]}

my_dict
```

Out[ ]:
```
{'course name': 'Intro to Python for Social Scientists',
 'starting_date': '2023.04.02',
 'num_participants': 20,
 'participant_names': ['Marko', 'Ana', 'Petar', 'Jovana']}
```

In [ ]:
```
## Dictionary with different datatypes 2 -> strings, integers, dictionaries, ...

dict_students = {"Ana" : "Acountant",
                 "Petar" : "Programmer",
                 "Marko" : "Manager",
                 "Jovana" : "Junior Developer"}

my_dict = {"course name" : "Intro to Python for Social Scientists",
           "starting_date" : "2023.04.02",
           "num_participants" : 20,
           "participant_names" : dict_students}

import pprint as pp
pp.pprint(my_dict)
```

```
{'course name': 'Intro to Python for Social Scientists',
 'num_participants': 20,
 'participant_names': {'Ana': 'Acountant',
                       'Jovana': 'Junior Developer',
                       'Marko': 'Manager',
                       'Petar': 'Programmer'},
 'starting_date': '2023.04.02'}
```

Keys 🔑🔑 and items 🚪

In [ ]:
```
my_dict.keys()
```

Out[ ]:
```
dict_keys(['course name', 'starting_date', 'num_participants', 'participant_names'])
```

In [ ]:
```
## Keys open the door to items 🚪
my_dict["course name"]
```

Out[ ]:
```
'Intro to Python for Social Scientists'
```

In [ ]:
```
my_dict.items()
```

Out[ ]:
```
dict_items([('course name', 'Intro to Python for Social Scientists'), ('starting_date',
'2023.04.02'), ('num_participants', 20), ('participant_names', {'Ana': 'Acountant', 'Pe
tar': 'Programmer', 'Marko': 'Manager', 'Jovana': 'Junior Developer'})])
```

Iterating over a dictionary

In [ ]:
```
## Iterating over keys:

for key in my_dict.keys():
    print(f"My key is currently: {key}. Let's see what items it opens:")
    print(f"---> {my_dict[key]}")
```

```
My key is currently: course name. Let's see what items it opens:
---> Intro to Python for Social Scientists
My key is currently: starting_date. Let's see what items it opens:
---> 2023.04.02
My key is currently: num_participants. Let's see what items it opens:
---> 20
My key is currently: participant_names. Let's see what items it opens:
---> {'Ana': 'Acountant', 'Petar': 'Programmer', 'Marko': 'Manager', 'Jovana': 'Junior
Developer'}
```

In [ ]:
```
## Iterating over keys and items:

for key, item in my_dict.items():
    print(key, "--->", item)
```

```
course name ---> Intro to Python for Social Scientists
starting_date ---> 2023.04.02
num_participants ---> 20
participant_names ---> {'Ana': 'Acountant', 'Petar': 'Programmer', 'Marko': 'Manager',
'Jovana': 'Junior Developer'}
```

In [ ]:
```
pets_list = ["Bobby", "Marley", "Lessie"]

pets_dict = {}
for pet in pets_list:
    pets_dict[pet] = "dog"

print(pets_dict)
```

```
{'Bobby': 'dog', 'Marley': 'dog', 'Lessie': 'dog'}
```

- Dictionary operations
- Dictionary VS other data types
- Why is a dictionary useful? It's the basis of Pandas `DataFrames` !

Introducing Pandas (very briefly)

We'll learn much more about Pandas in the following sessions.

It will become one of the main tools in our arsenal.

In this brief introduction to this library, we'll just build upon what we've learned about `dict` objects.

In [ ]:
```
import pandas as pd

countries_dict = {"country" : ["Serbia", "Slovenia", "Slowakia"],
                  "capital" : ["Belgrade", "Ljubljana", "Bratislava"],
                  "population" : [8000000, 3000000, 6000000]}

countries_df = pd.DataFrame(countries_dict)

countries_df
```

Out[ ]:

	country	capital	population
0	Serbia	Belgrade	8000000
1	Slovenia	Ljubljana	3000000
2	Slowakia	Bratislava	6000000

In [ ]:
```
countries_df["country"]
```

Out[ ]:
```
0      Serbia
1    Slovenia
2    Slowakia
Name: country, dtype: object
```

In [ ]:
```
countries_df["capital"]
```

Out[ ]:
```
0      Belgrade
1     Ljubljana
2    Bratislava
Name: capital, dtype: object
```

In [ ]:
```
countries_df["country"][:2]
```

Out[ ]:
```
0      Serbia
1    Slovenia
Name: country, dtype: object
```

That's enough about Pandas for today.

We will develop this knowledge much further (and deeper) in the next sessions.

`for` loops

In [ ]:
```
my_family = ["mum", "dad", "sister", "brother"]

for i in my_family:
        print("I love my", i)
```

```
I love my mum
I love my dad
I love my sister
I love my brother
```

In [ ]:
```
for letter in "family":
    print(letter)
```

```
f
a
m
i
l
y
```

In [ ]:
```
for letter in "family":
    print(letter * 3)
```

```
fff
aaa
mmm
iii
lll
yyy
```

In [ ]:
```
for i in range(10 + 1):
    print(i)
```

```
0
1
2
3
4
5
6
7
8
9
10
```

In [ ]:
```
## Let's have a closer look at the range() function

range(10)
```

Out[ ]:
```
range(0, 10)
```

Why did "nothing" happen?

In [ ]:
```
## Let's consult the expert on this one:

help(range)
```

```
Help on class range in module builtins:

class range(object)
 |  range(stop) -> range object
 |  range(start, stop[, step]) -> range object
 |
 |  Return an object that produces a sequence of integers from start (inclusive)
 |  to stop (exclusive) by step.  range(i, j) produces i, i+1, i+2, ..., j-1.
 |  start defaults to 0, and stop is omitted!  range(4) produces 0, 1, 2, 3.
 |  These are exactly the valid indices for a list of 4 elements.
 |  When step is given, it specifies the increment (or decrement).
 |
 |  Methods defined here:
 |
 |  __bool__(self, /)
 |      self != 0
 |
 |  __contains__(self, key, /)
 |      Return key in self.
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __ge__(self, value, /)
 |      Return self>=value.
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __getitem__(self, key, /)
 |      Return self[key].
 |
 |  __gt__(self, value, /)
 |      Return self>value.
 |
 |  __hash__(self, /)
 |      Return hash(self).
 |
 |  __iter__(self, /)
 |      Implement iter(self).
 |
 |  __le__(self, value, /)
 |      Return self<=value.
 |
 |  __len__(self, /)
 |      Return len(self).
 |
 |  __lt__(self, value, /)
 |      Return self<value.
 |
 |  __ne__(self, value, /)
 |      Return self!=value.
 |
 |  __reduce__(...)
 |      Helper for pickle.
 |
 |  __repr__(self, /)
 |      Return repr(self).
 |
```

```
|  __reversed__(...)
|      Return a reverse iterator.
|
|  count(...)
|      rangeobject.count(value) -> integer -- return number of occurrences of value
|
|  index(...)
|      rangeobject.index(value) -> integer -- return index of value.
|      Raise ValueError if the value is not present.
|
|  ----------------------------------------------------------------------
|  Static methods defined here:
|
|  __new__(*args, **kwargs) from builtins.type
|      Create and return a new object.  See help(type) for accurate signature.
|
|  ----------------------------------------------------------------------
|  Data descriptors defined here:
|
|  start
|
|  step
|
|  stop
```

So, the `range()` function on its own outputs a `range` object.

Let's doublecheck:

In [ ]:
```
type(range(10))
```

Out[ ]:
```
range
```

Most of the time, we'll need to pair this with another function to achieve our goals.

One such function is the `list()` function.

Can you guess the output of `list(range(11))` ?

In [ ]:
```
list(range(11))
```

Out[ ]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Let's learn a bit more about the `range()` function.

We've seen that it has three parameters: `range(start, stop[, step])` .

By defauls, `start = 0` , and `step = 1` .

In [ ]:
```
print(list(range(10)))
print(list(range(0, 10, 1))) ## Same as the above
print(list(range(2, 21, 2))) ## Now we're mixing it up a bit
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

Another typical use of the `range()` function is in combination with a for loop.

We instruct Python to perform a certain operation for every element generated within the `range` object.

Let's see some examples:

In [ ]:
```
for i in range(1, 11):
    if i % 2 == 0:
        print("We have an even number - it's", i)
    else:
        print("\tOdd one here", i)
```

```
        Odd one here 1
We have an even number - it's 2
        Odd one here 3
We have an even number - it's 4
        Odd one here 5
We have an even number - it's 6
        Odd one here 7
We have an even number - it's 8
        Odd one here 9
We have an even number - it's 10
```

🏠 Ask ChatGPT how we could make this code shorter!

🏠 For all integers between 1 and 100 (including 100), print "Woah" if the number is a multiplier of 3; print "Heey" if it's a multiplier of 5; print "Now that's special" if it's a multiplier of both 3 and 5; print the number itself if none of the above conditions are matcher.

Hint: use a for loop, then add some control flow :)

It can be implemented in different ways - think about the structure and elegance of the code.

`while` loops

In [ ]:
```
my_age = 12

while my_age < 18:
    print("I'm", my_age, "old. I'm not allowed to dring alcohol!")
    my_age += 1

print("Now I'm", my_age, "years old. Moderate drinking allowed!")
```

```
I'm 12 old. I'm not allowed to dring alcohol!
I'm 13 old. I'm not allowed to dring alcohol!
I'm 14 old. I'm not allowed to dring alcohol!
I'm 15 old. I'm not allowed to dring alcohol!
I'm 16 old. I'm not allowed to dring alcohol!
I'm 17 old. I'm not allowed to dring alcohol!
Now I'm 18 years old. Moderate drinking allowed!
```

Beware of never-ending loops!

Our loop needs to end - our condition must be such that it enables the completion of the operations!

Your computer: Please don't run this! my_bool = True while my_bool: print("This is not going to end (well)...")

Homework

🏠 Which objects are iterable in Python?

New data type: `tuple`

New data type: `set`