Introduction to

with Application to Bioinformatics

- Day 4

Start by doing today's quiz

Go to Canvas, `Modules -> Day 4 -> Review Day 3`

~20 minutes

In what ways does the type of an object matter?

- Questions 1, 2 and 3

In [ ]:
```
row = 'sofa|2000|buy|Uppsala'
fields = row.split('|')
price = fields[1]
if price == 2000:
    print('The price is a number!')
if price == '2000':
    print('The price is a string!')
```

In [ ]:
```
print(sorted([ 2000,   30,   100 ]))
```

In [ ]:
```
print(sorted(['2000', '30', '100']))
```

In what ways does the type of an object matter?

- Questions 1, 2 and 3

In [ ]:
```
row = 'sofa|2000|buy|Uppsala'
fields = row.split('|')
price = fields[1]
if price == 2000:
    print('The price is a number!')
if price == '2000':
    print('The price is a string!')
```

In [ ]:
```
print(sorted([ 2000,   30,   100 ]))
```

In [ ]:
```
print(sorted(['2000', '30', '100']))
```

In [ ]:
```
ord('2')
```

In what ways does the type of an object matter?

- Each type store a specific type of information
    - `int` for integers,
    - `float` for floating point values (decimals),
    - `str` for strings,
    - `list` for lists,
    - `dict` for dictionaries.
- Each type supports different operations, functions and methods.

- Each type supports different **operations**

In [ ]:
```
30 > 2000
```

In [ ]:
```
'30' > '2000'
```

In [ ]:
```
30 > int('2000')
```

In [ ]:
```
'12345'[2]
```

In [ ]:
```
12345[2]
```

- Each type supports different **functions**

In [ ]:
```
max('2000')
```

In [ ]:
```
max(2000)
```

In [ ]:
```
math.cos(3.14)
```

In [ ]:
```
math.cos('3.14')
```

- Each type supports different **methods**

In [ ]:
```
'ACTG'.lower()
```

In [ ]:
```
[1, 2, 3].lower()
```

In [ ]:
```
set([]).add('tiger')
```

In [ ]:
```
[].add('tiger')
```

- Each type supports different **methods**

In [ ]:
```
'ACTG'.lower()
```

In [ ]:
```
[1, 2, 3].lower()
```

In [ ]:
```
set([]).add('tiger')
```

In [ ]:
```
[].add('tiger')
```

- How to find what methods are available: Python documentation, or `dir()`

In [ ]:
```
dir('ACTG') # list all attributes
```

Convert string to number

- Questions 4, 5 and 6

In [ ]:
```
float('2000')
```

In [ ]:
```
float('0.5')
```

In [ ]:
```
float('1e9')
```

In [ ]:
```
float('1e-2')
```

In [ ]:
```
int('2000')
```

In [ ]:
```
int('1.5')
```

In [ ]:
```
int('1e9')
```

Convert to boolean: `1`, `0`, `'1'`, `'0'`, `''`, `{}`

- Question 7

In [ ]:
```
bool(1)
```

In [ ]:
```
bool(0)
```

In [ ]:
```
bool('1')
```

In [ ]:
```
bool('0')
```

In [ ]:
```
bool('')
```

In [ ]:
```
bool({})
```

- Python and the truth: true and false values

In [ ]:
```
values = [1, 0, '', '0', '1', [], [0]]
for x in values:
    if x:
        print(repr(x), 'is true!')
    else:
        print(repr(x), 'is false!')
```

- Python and the truth: true and false values

In [ ]:
```
values = [1, 0, '', '0', '1', [], [0]]
for x in values:
    if x:
        print(repr(x), 'is true!')
    else:
        print(repr(x), 'is false!')
```

- `if x` is equivalent to `if bool(x)`

- Is `1` equivalent to `True`?

In [ ]:
```
1 == True
```

In [ ]:
```
x = 1
if x is True:
    print(repr(x), 'is true!')
else:
    print(repr(x), 'is false!')
```

In [ ]:
```
x = 1
if bool(x) is True:
    print(repr(x), 'is true!')
else:
    print(repr(x), 'is false!')
```

- Is `1` equivalent to `True`?

In [ ]:
```
1 == True
```

In [ ]:
```
x = 1
if x is True:
    print(repr(x), 'is true!')
else:
    print(repr(x), 'is false!')
```

In [ ]:
```
x = 1
if bool(x) is True:
    print(repr(x), 'is true!')
else:
    print(repr(x), 'is false!')
```

- Be careful: `if x is True` is **not** equivalent to `if bool(x) is True`

Container types, when should you use which? (Question 8)

- **lists**: when order is important
- **dictionaries**: to keep track of the relation between keys and values
- **sets**: to check for membership. No order, no duplicates.

In [ ]:
```
genre_list = ["comedy", "drama", "drama", "sci-fi"]
genre_list
```

In [ ]:
```
genres = set(genre_list)
genres
```

In [ ]:
```
'drama' in genre_list
'drama' in genres
# which operation is faster?
```

In [ ]:
```
genre_counts = {"comedy": 1, "drama": 2, "sci-fi": 1}
genre_counts
```

In [ ]:
```
movie = {"rating": 10.0, "title": "Toy Story"}
movie
```

Python syntax (Question 9)

In [ ]:
```
def echo(message): # starts a new function definition
    # this function echos the message
    print(message) # print state of the variable
    return message # return the value to end the function
```

Converting between strings and lists

- Question 10

In [ ]:
```
list("hello")
```

In [ ]:
```
str(['h', 'e', 'l', 'l', 'o'])
```

In [ ]:
```
'_'.join(['h', 'e', 'l', 'l', 'o'])
```

What is a function?

- A named piece of code that performs a specific task
- A relation (mapping) between inputs (arguments) and output (return value)

In [ ]:

```
def increment_by_two(number):
    number += 2
    return number

print(increment_by_two(100))
```

TODAY

- More on functions:
    - scop of variables
    - positional arguments and keyword arguments
    - `return` statement
- Reusing code:
    - comments and documentation
    - importing modules: using libraries
- Pandas - explore your data!

More on functions: scope - global vs local variables

- Global variables can be accessed inside the function

In [ ]:

```
HOST = 'global'

def show_host():
    print(f'HOST inside the function = {HOST}')

show_host()
print(f'HOST outside the function = {HOST}')
```

- Change in the function will not change the global variable

In [ ]:

```
HOST = 'global'

def change_host():
    HOST = 'local'
    print(f'HOST inside the function = {HOST}')

print(f'HOST outside the function before change = {HOST}')
change_host()
print(f'HOST outside the function after change  = {HOST}')
```

- Pass global variable as argument

In [ ]:

```
HOST = 'global'

def change_host(HOST):
    HOST = 'local'
    print(f'HOST inside the function = {HOST}')

print(f'HOST outside the function before change = {HOST}')
change_host(HOST)
print(f'HOST outside the function after change  = {HOST}')
```

More on functions: scope - global vs local variables cont.

List as global variables

In [ ]:
```
MOVIES = ['Toy story', 'Home alone']

def change_movie():
    MOVIES = ['Fargo', 'The Usual Suspects']
    print(f'MOVIES inside the function = {MOVIES}')

print(f'MOVIES outside the function before change = {MOVIES}')
change_movie()
print(f'MOVIES outside the function after change  = {MOVIES}')
```

Will the global variable never to changed by function?

In [ ]:
```
MOVIES = ['Toy story', 'Home alone']

def change_movie():
    MOVIES.extend(['Fargo', 'The Usual Suspects'])
    print(f'MOVIES inside the function = {MOVIES}')

print(f'MOVIES outside the function before change = {MOVIES}')
change_movie()
print(f'MOVIES outside the function after change  = {MOVIES}')
```

Will the global variable never to changed by function?

In [ ]:
```
MOVIES = ['Toy story', 'Home alone']

def change_movie():
    MOVIES.extend(['Fargo', 'The Usual Suspects'])
    print(f'MOVIES inside the function = {MOVIES}')

print(f'MOVIES outside the function before change = {MOVIES}')
change_movie()
print(f'MOVIES outside the function after change  = {MOVIES}')
```

Take away: be careful when using global variables. Do not use it unless you know what you are doing.

More on functions: `return` statement

A function that counts the number of occurences of `'C'` in the argument string.

In [ ]:
```
def cytosine_count(nucleotides):
    count = 0
    for x in nucleotides:
        if x == 'c' or x == 'C':
            count += 1
    return count

count1 = cytosine_count('CATATTAC')
count2 = cytosine_count('tagtag')
print(count1, count2)
```

Functions that `return` are easier to repurpose than those that `print` their result

In [ ]:
```
cytosine_count('catattac') + cytosine_count('tactactac')
```

In [ ]:
```
def print_cytosine_count(nucleotides):
    count = 0
    for x in nucleotides:
        if x == 'c' or x == 'C':
            count += 1
    print(count)

print_cytosine_count('CATATTAC')
print_cytosine_count('tagtag')
```

In [ ]:
```
print_cytosine_count('catattac') + print_cytosine_count('tactactac')
```

- Functions without any `return` statement returns `None`

In [ ]:
```
def foo():
    do_nothing = 1

r = foo()
print(f'Return value of foo() = {r}')
```

- Functions without any `return` statement returns `None`

In [ ]:
```
def foo():
    do_nothing = 1

r = foo()
print(f'Return value of foo() = {r}')
```

- Use `return` for all values that you might want to use later in your program

Keyword arguments

In [ ]:
```
fh = open('files/recipes.txt',  mode='w', encoding='utf-8')
```

In [ ]:
```
sorted([1, 4, 100, 5, 6], reverse=True)
```

Why do we use keyword arguments?

In [ ]:
```
record = 'gene_id INSR "insulin receptor"'

record.split(' ', 2)
```

Why do we use keyword arguments?

In [ ]:
```
record = 'gene_id INSR "insulin receptor"'

record.split(' ', 2)
```

In [ ]:
```
record.split(sep=' ', maxsplit=2)
```

Why do we use keyword arguments?

In [ ]:
```
record = 'gene_id INSR "insulin receptor"'

record.split(' ', 2)
```

In [ ]:
```
record.split(sep=' ', maxsplit=2)
```

- It increases the clarity and redability

The order of keyword arguments does not matter

In [ ]:
```
fh = open('files/recipes.txt', mode='w', encoding='utf-8'); fh.close()
```

In [ ]:
```
fh = open('files/recipes.txt', encoding='utf-8', mode='w'); fh.close()
```

Can be used in both ways, with or without keyword

- if there is no ambiguity

In [ ]:
```
fh = open('files/recipes.txt', 'w', encoding='utf-8'); fh.close()
```

In [ ]:
```
fh = open('files/recipes.txt',  mode='w', encoding='utf-8'); fh.close()
```

But there are some exceptions

In [ ]:
```
fh = open('files/recipes.txt', encoding='utf-8', 'w'); fh.close()
```

But there are some exceptions

In [ ]:
```
fh = open('files/recipes.txt', encoding='utf-8', 'w'); fh.close()
```

- Positional arguments must be in front of keyword arguments

Restrictions by purpose

In [ ]:
```
sorted([1, 4, 100, 5, 6], reverse=True)
```

In [ ]:
```
sorted([1, 4, 100, 5, 6], True)
```

Restrictions by purpose

In [ ]:
```
sorted([1, 4, 100, 5, 6], reverse=True)
```

In [ ]:
```
sorted([1, 4, 100, 5, 6], True)
```

In [ ]:
```
sorted(iterable, /, *, key=None, reverse=False)
```

- arguments before `/` must be specified with position
- arguments after `*` must be specified with keyword

How to define functions taking keyword arguments

- Just define them as usual:

In [ ]:
```
def format_sentence(subject, value, end):
    return 'The ' + subject + ' is ' + value + end

print(format_sentence('lecture', 'ongoing', '.'))

print(format_sentence('lecture', 'ongoing', end='!'))

print(format_sentence(subject='lecture', value='ongoing', end='...'))
```

Defining functions with default arguments

In [ ]:
```
def format_sentence(subject, value, end='.'):
    return 'The ' + subject + ' is ' + value + end

print(format_sentence('lecture', 'ongoing'))

print(format_sentence('lecture', 'ongoing', '...'))
```

Defining functions with optional arguments

- Convention: use the object `None`

In [ ]:
```
def format_sentence(subject, value, end='.', second_value=None):
    if second_value is None:
        return 'The ' + subject + ' is ' + value + end
    else:
        return 'The ' + subject + ' is ' + value + ' and ' + second_val

print(format_sentence('lecture', 'ongoing'))

print(format_sentence('lecture', 'ongoing',
                      second_value='self-referential', end='!'))
```

Small detour: Python's value for missing values: `None`

- Default value for optional arguments
- Implicit return value of functions without a `return` statement
- `None` is `None`, not anything else

In [ ]:
```
None == 0
```

In [ ]:
```
None == False
```

In [ ]:
```
None == ''
```

In [ ]:
```
bool(None)
```

In [ ]:
```
type(None)
```

Exercise 1

- Notebook Day_4_Exercise_1 (~30 minutes)
- Go to Canvas, `Modules -> Day 4 -> Exercise 1 - day 4`
- Quiz. Go to Canvas, `Modules -> Day 4 -> PyQuiz 4.1`
- Lunch break
- Extra reading:
    - https://realpython.com/python-kwargs-and-args/
    - https://able.bio/rhett/python-functions-and-best-practices--78aclaa

A short note on code structure

- Functions
    - e.g. sum(), print(), open()
- Modules
    - files containing a collection of functions and methods, e.g. string.py
- Documentation
    - docstring, comments

Why functions?

- Cleaner code
- Better defined tasks in code
- Re-usability
- Better structure

Why modules?

- Cleaner code
- Better defined tasks in code
- Re-usability
- Better structure

Why modules?

- Cleaner code
- Better defined tasks in code
- Re-usability
- Better structure

- Collect all related functions in one file
- Import a module to use its functions
- Only need to understand what the functions do, not how

Example of modules

In [ ]:
```
import sys

sys.argv[1]
```

In [ ]:
```
from datetime import datetime
print(datetime.now())
```

In [ ]:
```
import os

os.system("date")
```

How to find the right module and instructions?

- Look at the module index for Python standard modules
- Search PyPI
- Search https://www.w3schools.com/python/
- Ask your colleagues
- Search the web
- Use ChatGPT

How to find the right module and instructions?

- Look at the module index for Python standard modules
- Search PyPI
- Search https://www.w3schools.com/python/
- Ask your colleagues
- Search the web
- Use ChatGPT

- Standard modules: no installation needed
- Other libraries: install with `pip install` or `conda install`

How to understand it?

- E.g. I want to know how to split a string by the separator `,`

In [ ]:
```
text = 'Programming,is,cool'
```

How to understand it?

- E.g. I want to know how to split a string by the separator `,`

In [ ]:
```
text = 'Programming,is,cool'
```

In [ ]:
```
help(text.split)
```

How to understand it?

- E.g. I want to know how to split a string by the separator `,`

In [ ]:
```
text = 'Programming,is,cool'
```

In [ ]:
```
help(text.split)
```

In [ ]:
```
text.split(sep=',')
```

For slightly more complicated problems

- e.g. how to download Python logo from internet with `urllib`, given the URL https://www.python.org/static/img/python-logo@2x.png

In [ ]:
```
import urllib

help(urllib)
```

For slightly more complicated problems

- e.g. how to download Python logo from internet with `urllib`, given the URL https://www.python.org/static/img/python-logo@2x.png

In [ ]:
```
import urllib

help(urllib)
```

- Probably easier to find the answer by searching the web or using ChatGPT

One minute exercise

- get help from ChatGPT (https://chat.openai.com/)

Using Python to download the Python logo from internet with urllib providing the url as https://www.python.org/static/img/python-logo@2x.png

One minute exercise

- get help from ChatGPT (https://chat.openai.com/)

Using Python to download the Python logo from internet with urllib providing the url as https://www.python.org/static/img/python-logo@2x.png

In [ ]:

```
import urllib.request

url = "https://www.python.org/static/img/python-logo@2x.png"
filename = "python-logo.png"  # The name you want to give to the downlc

urllib.request.urlretrieve(url, filename)

print("Download completed.")
```

Documentation and commenting your code

Remember `help()`?

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

Documentation and commenting your code

Remember `help()`?

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

- This works because somebody else has documented their code!

Documentation and commenting your code

Remember `help()`?

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

- This works because somebody else has documented their code!

In [ ]:
```
def process_file(filename, chrom, pos):
    """
    Read a vcf file, search for lines matching
    chromosome chrom and position pos.

    Print the genotypes of the matching lines.
    """
    for line in open(filename):
        if not line.startswith('#'):
            col = line.split('\t')
            if col[0] == chrom and int(col[1]) == pos:
                print(col[9:])
```

Your code may have two types of users:

- library users
- maintainers (maybe yourself!)

Your code may have two types of users:

- library users
- maintainers (maybe yourself!)

Write documentation for both of them!

- library users (docstrings):

```
"""
What does this function do?
"""
```

- maintainers (comments):

```
# implementation details
```

Places for documentation

- At the beginning of the file

```
"""
  This module provides functions for ...
  """
```

- At every function definition

In [ ]:

```
import random
def make_list(x):
    """Returns a random list of length x."""
    li = list(range(x))
    random.shuffle(li)
    return li
```

Comments

- Wherever the code is hard to understand

In [ ]:
```
my_list[5] += other_list[3]  # explain why you do this!
```

Demo: write a Python script with documentation and use it

In [ ]:

Read more:

https://realpython.com/documenting-python-code/

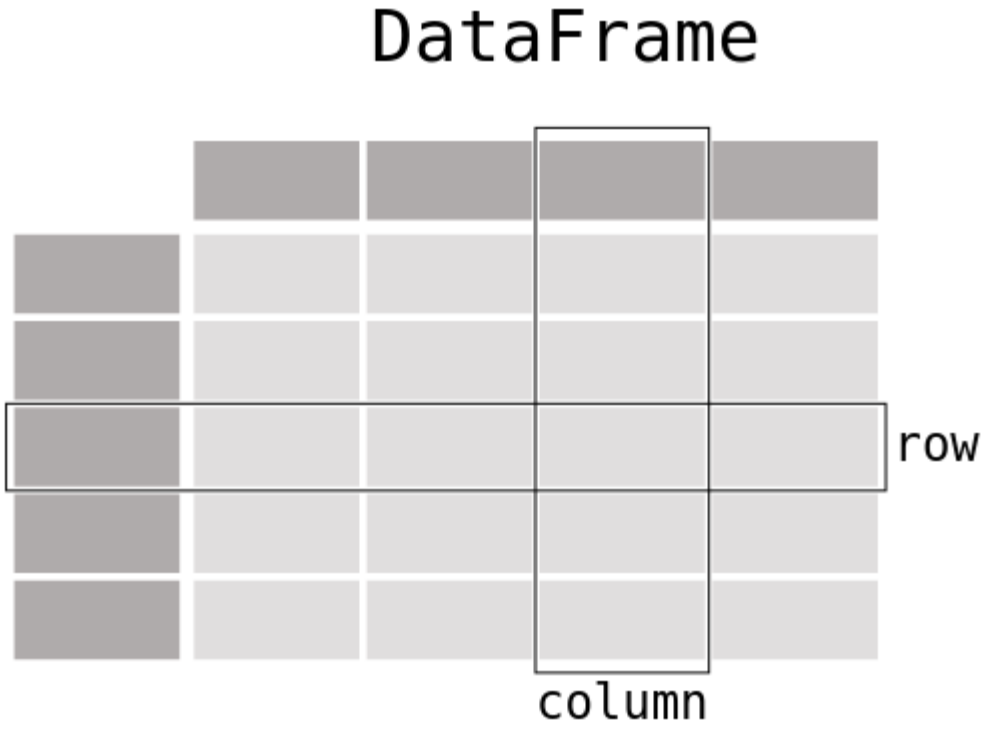
https://www.python.org/dev/peps/pep-0008/?#comments

Pandas!!!

Pandas

- Library for working with tabular data
- Data analysis:
    - filter
    - transform
    - aggregate
    - plot
- Main hero: the `DataFrame` type

DataFrame



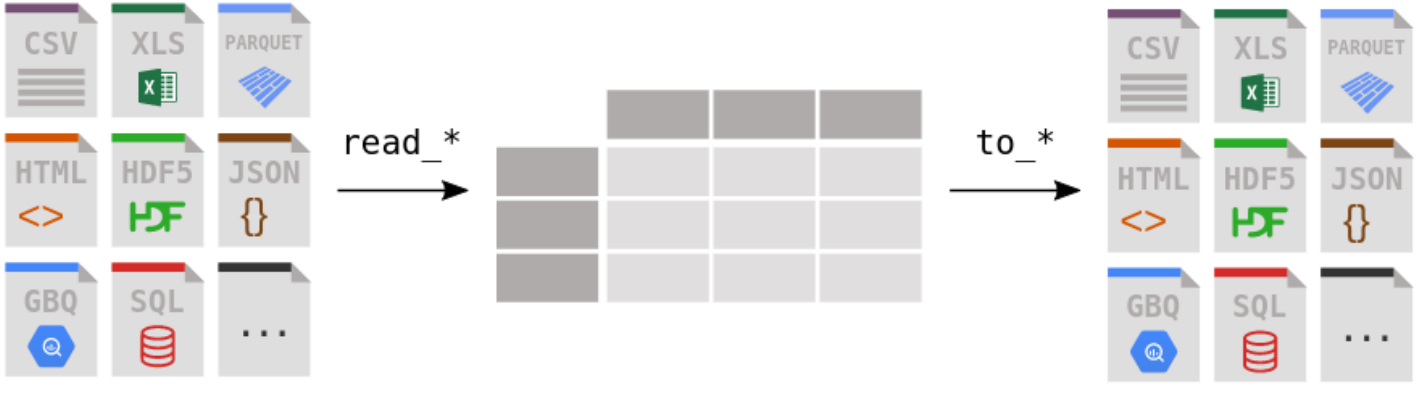
Creating a small DataFrame

In [ ]:

```
import pandas as pd
data = {
    'age': [1,2,3,4],
    'circumference': [2,3,5,10],
    'height': [30, 35, 40, 50]
}
df = pd.DataFrame(data)
df
```

Pandas can import data from many formats

- `pd.read_table` : tab separated values `.tsv`
- `pd.read_csv` : comma separated values `.csv`
- `pd.read_excel` : Excel spreadsheets `.xlsx`
- For a data frame `df` : `df.to_table()` , `df.to_csv()` , `df.to_excel()`



Orange tree data

In [ ]:
```
!cat ../downloads/Orange_1.tsv
```

In [ ]:
```
df = pd.read_table('../downloads/Orange_1.tsv')
df
```

Orange tree data

In [ ]:
```
!cat ../downloads/Orange_1.tsv
```

In [ ]:
```
df = pd.read_table('../downloads/Orange_1.tsv')
df
```

- One implict index (0, 1, 2, 3)
- Columns: `age`, `circumference`, `height`
- Rows: one per data point, identified by their index

Read data from Excel file

In [ ]:
```
df2 = pd.read_excel('../downloads/Orange_1.xlsx')
df2
```

Overview of your data, basic statistics

In [ ]:
```
df
```

In [ ]:
```
df.shape
```

In [ ]:
```
df.describe()
```

In [ ]:
```
df.std()
```

Selecting columns from a dataframe

```
dataframe.columnname
dataframe['columnname']
```

Selecting one column

In [ ]:
```
df
```

In [ ]:
```
df.age
```

In [ ]:
```
df['age']
```

Selecting multiple columns

In [ ]:
```
df
```

In [ ]:
```
df[['age', 'height']]
```

Selecting multiple columns

In [ ]:
```
df
```

In [ ]:
```
df[['age', 'height']]
```

In [ ]:
```
df[['height', 'age']] # what's the difference?
```

Selecting rows from a dataframe

In [ ]:
```
df
```

In [ ]:
```
df.loc[0] # select the first row
```

In [ ]:
```
df.loc[1:3] # select from row 2 to 4
```

In [ ]:
```
df.loc[[1, 3]] # select row 2 and 4
```

Selecting cells from a dataframe

In [ ]:
```
df
```

In [ ]:
```
df.loc[[1, 3], ['age', 'height']]
```

Run statistics on specific rows, columns, cells

In [ ]:
```
df[['age', 'circumference']].describe()
```

In [ ]:
```
df['age'].std()
```

In [ ]:

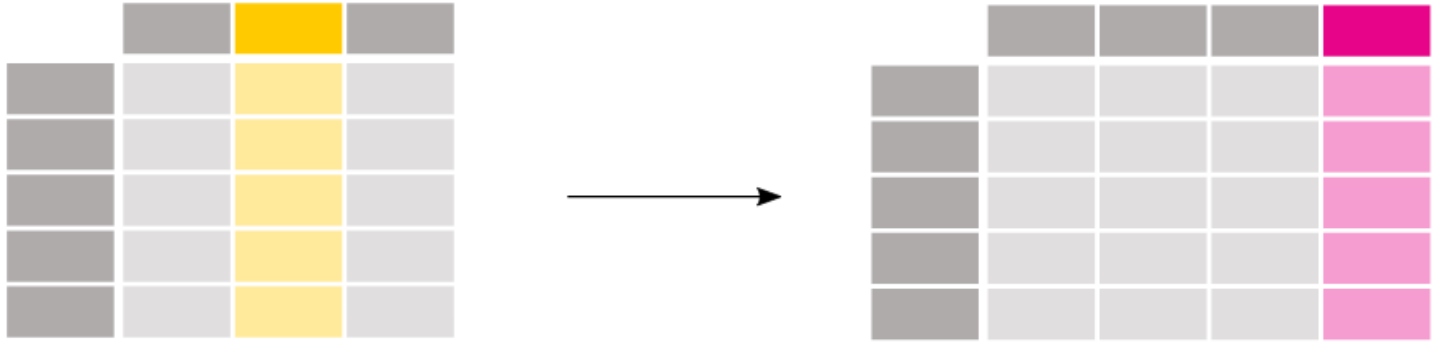
```
df.loc[1:3, ['age']].mean()
```

Selecting data from a dataframe by index

```
dataframe.iloc[index]
dataframe.iloc[start:stop]
```

Further reading from pandas documentation:
https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.iloc.html

Creating new column derived from existing column



In [ ]:

```
import math
df['radius'] = df['circumference'] / 2.0 / math.pi
df
```

Expand dataframe by concatenating

In [ ]:
```
df1 = pd.DataFrame({
    'age': [1,2,3,4],
    'circumference': [2,3,5,10],
    'height': [30, 35, 40, 50]
})
```

In [ ]:
```
df2 = pd.DataFrame({
    'name': ['palm', 'ada', 'ek', 'olive'],
    'price': [1423, 2000, 102, 30]
})
```

In [ ]:
```
df1
```

In [ ]:
```
df2
```

In [ ]:

```
pd.concat([df1, df2], axis=1)
```

Selecting/filtering the dataframe by condition

e.g.

- Only trees with age larger than 100
- Only tree with circumference shorter than 20

Slightly bigger data frame of orange trees

In [ ]:
```
!head -n 10 ../downloads/Orange.tsv
```

In [ ]:
```
df = pd.read_table('../downloads/Orange.tsv')
df.iloc[0:5]  # can also use .head()
```

In [ ]:
```
df.Tree.unique()
```

Selecting with condition

In [ ]:
```
df[df['Tree'] == 1]
```

In [ ]:
```
df[df.age > 500]
```

In [ ]:
```
df[(df.age > 500) & (df.circumference < 100)]
```

Small exercise 1

- find the maximal circumference and then filter the data frame by it

In [ ]:
```
df[df.circumference == df.circumference.max() ]
```

Small exercise 2

Here's a dictionary of students and their grades:

```
students = {'student': ['bob', 'sam', 'joe'], 'grade': [1, 3, 4]}
```

Use Pandas to:

- create a dataframe with this information
- get the mean value of the grades

In [ ]:

```
```

Plotting

```
df.columnname.plot()
```

Plotting

```
df.columnname.plot()
```

In [ ]:

```
small_df = pd.read_table('../downloads/Orange_1.tsv')
small_df
```

In [ ]:

```
small_df.plot(x='age', y='height', kind='line') # plot the relationship
# try with other types of plots, e.g. scatter
```

What if no plots shows up?

In [ ]:
```
import matplotlib.pyplot as plt
plt.show()
```

In [ ]:
```
%matplotlib inline
```

Plotting - bars

In [ ]:
```
small_df[['age']].plot(kind='bar')
```

Plotting multiple columns

In [ ]:
```
small_df[['circumference', 'age']].plot(kind='bar')
```

Plotting histogram

In [ ]:
```
small_df.plot(kind='hist', y = 'age')
```

Plotting box

In [ ]:
```
small_df.plot(kind='box', y = 'age')
```

Further reading: https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.plot.html

Exercise 2 (~30 minutes)

- Go to Canvas, `Modules -> Day 4 -> Exercise 2 - day 4`
- **Easy**:
    - Explore the `Orange_1.tsv`
- **Medium/hard**:
    - Use Pandas to read IMDB
    - Explore it by making graphs
- **Extra exercises**:
    - Read the pandas documentation :)
    - Start exploring your own data
- After exercise, do Quiz 4.2 and then take a break
- After break, working on the project