Introduction to

with Application to Bioinformatics

- Day 4

TODAY

- Keyword arguments
- Loops with break and continue
- "Code structure": comments and documentation
- Importing modules: using libraries
- Pandas - explore your data!

Start by doing today's quiz (Review Day 3)

Review: In what ways does the type of an object matter?

In [ ]:
```
row = 'sofa|2000|buy|Uppsala'
fields = row.split('|')
price = int(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 ]))
print(sorted(['2000', '30', '100']))
# Hint: is `'30' > '2000'`?
```

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**, functions and methods

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

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

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

- Each type supports different operations, functions and **methods**

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

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

- Convert to number: `'2000'` and `'0.5'` and `'1e9'`

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

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

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

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

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

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

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

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

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!')
```

- Converting between strings and lists

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

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

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

Container types, when should you use which?

- **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)
'drama' in genres
```

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

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

What is a function?

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

```
def hello_function(number):
    # print the user input
    print(number)
    number += 2
    return 2
```

More on functions

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('tactactac')
```

- Objects and references to objects

In [ ]:
```
list_A = ['red', 'green']
list_B = ['red', 'green']
list_B.append('blue')
print(list_A, list_B)
```

In [ ]:
```
list_A = ['red', 'green']
list_B = list_A           # another name to the SAME list. Aliasing
list_B.append('blue')
print(list_A, list_B)
```

In [ ]:
```
list_A = ['red', 'green']
list_B = list_A
list_A = []
print(list_A, list_B)
```

- Objects and references to objects, cont.

In [ ]:
```
list_A = ['red', 'green']
lists = {'A': list_A, 'B': list_A}
print(lists)
lists['B'].append('blue')
print(lists)
```

In [ ]:
```
list_A = ['red', 'green']
lists = {'A': list_A, 'B': list_A}
print(lists)
lists['B'] = lists['B'] + ['yellow']
print(lists)
```

Scope: global variables and local function variables

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

In [ ]:
```
def some_thriller_movies():
    return ['Fargo', 'The Usual Suspects']

movies = some_thriller_movies()
print(movies)
```

In [ ]:
```
def change_to_drama(movies):
    movies = ['Forrest Gump', 'Titanic']

change_to_drama(movies)
print(movies)
```

Keyword arguments

- A way to give a name explicitly to a function for clarity

In [ ]:
```
sorted(list('file'), reverse=True)
```

In [ ]:
```
attribute = 'gene_id "unknown gene"'
attribute.split(sep=' ', maxsplit=1)
```

In [ ]:
```
# print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
print('x=', end='')
print('1')
```

Keyword arguments

- Order of keyword arguments do not matter

```
open(file, mode='r', encoding=None) # some arguments omitted
```

- These mean the same:

```
open('files/recipes.txt', 'w', encoding='utf-8')
open('files/recipes.txt', mode='w', encoding='utf-8')
open('files/recipes.txt', encoding='utf-8', mode='w')
```

Defining 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='...'))
```

In [ ]:
```
print(format_sentence(subject='lecture', 'ongoing', '.'))
```

- Positional arguments comes first, keyword arguments after!

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_value + end

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`
- Something to initialize variable with no value yet
- Argument to a function indicating use the default value

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

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

Comparing None

- To differentiate `None` to the other false values such as `0`, `False` and `''` use `is None`:

In [ ]:
```
counts = {'drama': 2, 'romance': 0}

counts.get('romance'), counts.get('thriller')
```

In [ ]:
```
counts.get('romance') is None
```

In [ ]:
```
counts.get('thriller') is None
```

- Python and the truth, take two

In [ ]:
```
values = [None, 1, 0, '', '0', '1', [], [0]]
for x in values:
    if x is None:
        print(repr(x), 'is None')
    if not x:
        print(repr(x), 'is false')
    if x:
        print(repr(x), 'is true')
```

Controlling loops - `break`

```
for x in lines_in_a_big_file:
    if x.startswith('>'):  # this is the only line I want!
        do_something(x)
```

Controlling loops - `break`

```
for x in lines_in_a_big_file:
    if x.startswith('>'):  # this is the only line I want!
        do_something(x)
```

...waste of time!

Controlling loops - `break`

```
for x in lines_in_a_big_file:
    if x.startswith('>'):  # this is the only line I want!
        do_something(x)
```

...waste of time!

```
for x in lines_in_a_big_file:
    if x.startswith('>'):  # this is the only line I want!
        do_something(x)
        break  # break the loop
```

break

```
for line in file:
    if line.startswith('#'):
        break
    do_something(line)


print("I am done")
```

Controlling loops - `continue`

```
for x in lines_in_a_big_file:
    if x.startswith('>'):  # irrelevant line
        # just skip this! don't do anything
    do_something(x)
```

Controlling loops - `continue`

```
for x in lines_in_a_big_file:
    if x.startswith('>'):  # irrelevant line
        # just skip this! don't do anything
    do_something(x)
```

```
for x in lines_in_a_big_file:
    if x.startswith('>'):  # irrelevant line
        continue  # go on to the next iteration
    do_something(x)
```

Controlling loops - `continue`

```
for x in lines_in_a_big_file:
    if x.startswith('>'):  # irrelevant line
        # just skip this! don't do anything
    do_something(x)
```

```
for x in lines_in_a_big_file:
    if x.startswith('>'):  # irrelevant line
        continue  # go on to the next iteration
    do_something(x)
```

```
for x in lines_in_a_big_file:
    if not x.startswith('>'):  # not irrelevant!
        do_something(x)
```

continue

```
for line in file:
    if line.startswith('#'):
        continue
    do_something(line)


print("I am done")
```

Another control statement: pass - the placeholder

In [ ]:
```
def a_function():
    # I have not implemented this just yet
```

In [ ]:
```
def a_function():
    # I have not implemented this just yet
    pass
a_function()
```

Exercise 1

- Notebook Day_4_Exercise_1 (~30 minutes)

A short note on code structure

- functions
- modules (files)
- documentation

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: sys

```
import sys

sys.argv[1]
```

or

```
import pprint
pprint.pprint(a_big_dictionary)
```

Python standard modules

Check out the module index (https://docs.python.org/3.6/py-modindex.html)

How to find the right module?

How to understand it?

How to find the right module?

- look at the module index
- search PyPI (http://pypi.org)
- ask your colleagues
- search the web!

How to understand it?

How to understand it?

In [ ]:
```
import math

help(math.acosh)
```

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

In [ ]:

```
help(math.sqrt)

# install packages using: pip
# Dimitris' protip: install packages using conda
```

In [ ]:
```
math.sqrt(3)
```

Importing

In [ ]:
```
import math

math.sqrt(3)
```

Importing

In [ ]:
```
import math

math.sqrt(3)
```

In [ ]:
```
import math as m
m.sqrt(3)
```

Importing

In [ ]:
```
import math

math.sqrt(3)
```

In [ ]:
```
import math as m
m.sqrt(3)
```

In [ ]:
```
from math import sqrt
sqrt(3)
```

Documentation and commenting your code

Remember `help()`?

Works because somebody else has documented their code!

Documentation and commenting your code

Remember `help()`?

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 col[1] == pos:
                print(col[9:])
help(process_file)
```

Documentation and commenting your code

Remember `help()`?

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 col[1] == pos:
                print(col[9:])
help(process_file)
```

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

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
```

Documentation:

- At the beginning of the file

```
"""
 This module provides functions fo
r...
 """
```

- For every function

```
def make_list(x):
      """Returns a random list of length
 x."""
      pass
```

Comments:

- Wherever the code is hard to understand

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

Read more:

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

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

Formatting

In [ ]:
```
title = 'Toy Story'
rating = 10
print('The result is: ' + title + ' with rating: ' + str(rating))
```

In [ ]:
```
# f-strings (since python 3.6)
print(f'The result is: {title} with rating: {rating}')
```

In [ ]:
```
# format method
print('The result is: {} with rating: {}'.format(title, rating))
```

In [ ]:
```
# the ancient way (python 2)
print('The result is: %s with rating: %s' % (title, rating))
```

Learn more from the Python docs: https://docs.python.org/3.9/library/string.html#format-string-syntax (https://docs.python.org/3.9/library/string.html#format-string-syntax)

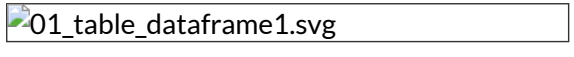
Exercise 2

Documentation

- Notebook Day_4_Exercise_2

Pandas

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



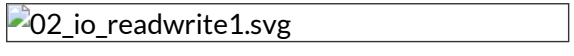
Creating a small DataFrame

In [ ]:

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

Selecting columns from a dataframe

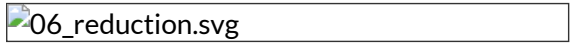
```
dataframe.columnname
dataframe['columnname']
```

In [ ]:
```
df.columns
```

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

In [ ]:
```
df.height
```

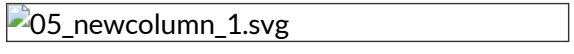
Calculating aggregated summary statistics



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

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

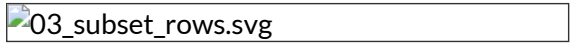
Creating new column derived from existing column



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

Selecting rows from a dataframe by index

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



In [ ]:
```
df.iloc[1:3]
```

Slightly bigger data frame of orange trees

In [ ]:
```
!head -n 6 ../downloads/Orange.tsv
```

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

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

In [ ]:
```
type(pd.DataFrame({"genre": ['Thriller', 'Drama'], "rating": [10, 9]}).rating.iloc[0])
```

In [ ]:

```
#young = df[df.age < 200]
#young
df[df.age < 1000]
```

Finding the maximum and then filter by it

```
df.loc[ df.age < 200 ]
```

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

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

In [ ]:
```
df[df.circumference == max_c]
```

Plotting

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

Plotting

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

In [ ]:
```
small_df = pd.read_table('../downloads/Orange_1.tsv')
small_df.plot(x='age', y='height')
```

Plotting

What if no plot shows up?

```
%pylab inline   # jupyter notebooks
```

or

```
import matplotlib.plot as plt

plt.show()
```

Plotting - many trees

- Plot a bar chart

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

In [ ]:
```
df[['circumference', 'age']].plot.bar(figsize=(12, 8), fontsize=16)
```

Scatterplot

```
df.plot.scatter(x="column_name", y="other_column_name")
```

In [ ]:
```
df.plot.scatter(x='age', y='circumference',
        figsize=(12, 8), fontsize=14)
```

Line plot

```
dataframe.plot.line(x=..., y=...)
```

In [ ]:
```
tree1 = df[df['Tree'] == 1]
tree1.plot.line(x='age', y='circumference',
           fontsize=14, figsize=(12,8))
```

In [ ]:
```
df.groupby('Tree').plot.line(x='age', y='circumference')
```

Exercise 3

- Read the `Orange_1.tsv`
  - Print the height column
  - Print the data for the tree at age 2
  - Find the maximum circumference
  - What tree reached that circumference, and how old was it at that time?
- Use Pandas to read IMDB
  - Explore it by making graphs
- Extra exercises:
  - Read the pandas documentation :)
  - Look at seaborn for a more feature-rich plotting lib