Make your analyses smarter
ORGANIZATION
Lecture: Friday, 09:30 - 11:00
Exercise: Friday, 12:15 - 14:30
Tools: Zoom, Jitsi Meet @ LRZ
Schedule
<table>
<thead>
<tr>
<th>Date</th>
<th>Topic</th>
<th>Date</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>May 1st</td>
<td>Holiday</td>
<td>Jun 12th</td>
<td>NoSQL DBs</td>
</tr>
<tr>
<td>May 8th</td>
<td>Intro/Basics</td>
<td>Jun 19th</td>
<td>ML intro</td>
</tr>
<tr>
<td>May 15th</td>
<td>Pythonics</td>
<td>Jun 26th</td>
<td>Visualization</td>
</tr>
<tr>
<td>May 22nd</td>
<td>Vectorization</td>
<td>Jul 3rd</td>
<td>Interactive I</td>
</tr>
<tr>
<td>May 29th</td>
<td>Biological DBs</td>
<td>Jul 10th</td>
<td>Interactive II</td>
</tr>
<tr>
<td>Jun 5th</td>
<td>Pandas</td>
<td>Jul 17th</td>
<td>Recap</td>
</tr>
</tbody>
</table>

subject to change
If there is no lecture the exercise takes place one week later

- Exercises are published on Fridays and discussed Friday the week after
- Last sheet/exercise: Jul 3rd / Jul 10th
- Exam (working date): Jul 24th, to be discussed
First Iteration

Experience driven syllabus - subject to change

Depending on the progress in the lecture single topics could be added or dropped

The sequence of topics might be shuffled

Hybrid nature: presentation of theoretical concepts are blended with back- and front-end technology
Exercises help to convert knowledge into a skill
Practical application of topics covered in the lecture
Active exploration of bioinformatics resources
Explore visualizations with Python and JavaScript
Use Python as common platform
Exercise Structure

- Rehearsing and explaining lecture material
- Discussion of previous homework
- Q&A session
- Introduction to new homework
Present topics beyond the canonical syllabus which are useful in Bioinformatics research

- Efficient data handling with Python
- Biological databases
- Current techniques and methods for data assembling and processing
- Visualizing results of data analysis
- Building and deploying predictive models
Python:
- Syntax
- Data types and operations on them
- Control structures
- Functions
- Modules
- Objects
- Idiomatic & Efficient Python
Efficient Data Handling:

- Faster data wrangling with NumPy arrays - explaining memory consumption and vectorized computations
- NumPy API through examples
- Scientific computing with SciPy
- Pandas dataframes
Biological Databases:
- Database Taxonomy: Primary and Secondary DBs
- Where to obtain data? Genbank, UniProt & PDB
- Accessing biological databases (through a web interface and programmatically)
- Parsing biological data (manually and through BioPython)
- Assembling a dataset from biological data

Datasets:
- Feature processing and engineering
- Visualizing analyses with Matplotlib and Seaborn
- Building predictive models with Scikit-Learn
Topic III: Publishing a Model

» Publishing a novel method
  » Compiling Python code with Cython and Numba
  » Packaging a model for the Web
  » Making a model available online: Flask and MongoDB
  » Enriching the online UI with JavaScript and D3.JS
PYTHON: OVERVIEW
We start with a shallow and narrow overview of Python. We will cover the following:

- Variables
- Basic data types
- Control structures
- Functions
- Classes
- Modules
Python is a strongly typed dynamic language.

<table>
<thead>
<tr>
<th>my_variable = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>my_variable = 'one'</td>
</tr>
<tr>
<td>print(my_variable)</td>
</tr>
<tr>
<td>one</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>my_variable = 1 + 'one'</th>
</tr>
</thead>
<tbody>
<tr>
<td>Traceback (most recent call last):</td>
</tr>
<tr>
<td>File &quot;&lt;stdin&gt;&quot;, line 1, in &lt;module&gt;</td>
</tr>
<tr>
<td>TypeError: unsupported operand type(s) for +: 'int' and 'str'</td>
</tr>
</tbody>
</table>
Variable names can contain letters, digits, and the underscore symbol. Variable names can’t start with a digit:

```
myvar1 = 1
my_var_1 = 'one'
_myvar1 = 'and another one'
1myvar = 'illegal'
```

File "<stdin>", line 1
```
  1myvar = 'illegal'
```
SyntaxError: invalid syntax
Python: Variables

Python keywords can’t be used as variable names. We can get a list of keywords by running the following code:

```python
import keyword
print(keyword.kwlist)
```

```
['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']
```
Use lowercase for the variable names, with words separated by underscores:

```python
use_this = True
doNotUseThis = True
```
The basic data types we are going to cover today are:

- `int` for integral numbers
- `float` for floating-point numbers
- `bool` for Boolean values
- `str` for strings
- `list`, `tuple`, and `range` for sequences
- `set` for sets
- `dict` for mappings
Python has the usual control flow constructs, such as branches and loops. Branching is achieved via an `if` statement. There is no `switch`-like statement in Python. `for` and `while` statements allow us to repeatedly execute a block of code. There is no `do_while` statement in Python.
`int` is a built-in integral data type. The range of possible values is limited only by the machine’s memory:

\[
\begin{array}{|c|}
\hline
9999999999999999999 
\times 2 \\
19999999999999999998 \\
\hline
\end{array}
\]
We can compare integers using comparison operators:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 &gt; 3</td>
<td>True</td>
</tr>
<tr>
<td>5 &gt;= 3</td>
<td>True</td>
</tr>
<tr>
<td>5 != 3</td>
<td>True</td>
</tr>
<tr>
<td>5 &lt; 3</td>
<td>False</td>
</tr>
<tr>
<td>5 &lt;= 3</td>
<td>False</td>
</tr>
<tr>
<td>5 == 3</td>
<td>False</td>
</tr>
</tbody>
</table>
True and False are Boolean values. We can use them together with *if* statement to perform branching, that is, execute a block of code if a condition is true:

```python
x = 6
if x < 0:
    print('Negative')
if x > 0:
    print('Positive')

Positive
```
We can add another code branch using `else` statement. The block of code associated with `else` statement is executed if a condition is false:

```python
x = 6
if x < 0:
    print('Negative')
else:
    print('Positive')
```

Positive
We can add even more branches using `elif` statement:

```python
x = 0
if x < 0:
    print('Negative')
elif x == 0:
    print('Zero')
else:
    print('Positive')
Zero
```
### Data Types: int

We can perform arithmetic operations on integers via arithmetic operators:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>$5 + 3$</td>
<td>8</td>
</tr>
<tr>
<td>$5 - 3$</td>
<td>2</td>
</tr>
<tr>
<td>$5 \times 3$</td>
<td>15</td>
</tr>
<tr>
<td>$5 / 3$</td>
<td>1.666...667</td>
</tr>
<tr>
<td>$5 \div 3$</td>
<td>1</td>
</tr>
<tr>
<td>$5 \mod 3$</td>
<td>2</td>
</tr>
<tr>
<td>$5 ** 3$</td>
<td>125</td>
</tr>
<tr>
<td>$-5$</td>
<td>-5</td>
</tr>
</tbody>
</table>
An acronym PEMDAS (parentheses, exponentiation, multiplication, division, addition, subtraction) helps to remember precedence of the operations:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>(3 + 2) * 4 / 2 ** 2 + 1</td>
<td>6.0</td>
</tr>
<tr>
<td>7 + 6 - 5 * 4 / 3 ** (2 + 1)</td>
<td>12.25925925925926</td>
</tr>
</tbody>
</table>
**Data Types: float**

`float` is a built-in type for floating-point values. These values have limited precision:

<table>
<thead>
<tr>
<th>0.1 == 0.10000000000000001</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
</tr>
</tbody>
</table>

We can use a decimal point or exponential notation to create floating-point values:

<table>
<thead>
<tr>
<th>12e3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.012</td>
</tr>
</tbody>
</table>
We can also use comparison and arithmetic operators with floats:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
<th>Operation</th>
<th>Result</th>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 &gt; 3.2</td>
<td>True</td>
<td>5.1 + 3.2</td>
<td>8.3</td>
<td>5.1 / 3.2</td>
<td>1.593...998</td>
</tr>
<tr>
<td>5.1 &lt; 3.2</td>
<td>False</td>
<td>5.1 - 3.2</td>
<td>1.899...995</td>
<td>5.1 // 3.2</td>
<td>1.0</td>
</tr>
<tr>
<td>5.1 != 3.2</td>
<td>True</td>
<td>5.1 * 3.2</td>
<td>16.32</td>
<td>5.1 ** 3.2</td>
<td>183.74...95</td>
</tr>
</tbody>
</table>
We have already seen True and False, built-in Boolean values. We can perform logical operations using the following operators:

<table>
<thead>
<tr>
<th>True  and False</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>True  or False</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>not True</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
</tr>
</tbody>
</table>
Boolean values are evaluated as 1 and 0 when used with arithmetic operators:

<table>
<thead>
<tr>
<th>True + 1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 - False</td>
<td>5</td>
</tr>
</tbody>
</table>
while loop executes a block of code as long as the condition is true:

```python
x = 0
while x < 5:
    print(x)
    x = x + 1
```

0
1
2
3
4
The null object, named None, is used to represent the absence of a value.
Python has a data type for strings (sequences of characters) and doesn’t have a specific data type for a single character, although we can create a string containing only one character (or an empty string containing zero characters):

<table>
<thead>
<tr>
<th>'Hello, World!'</th>
</tr>
</thead>
<tbody>
<tr>
<td>'Hello, World!'</td>
</tr>
<tr>
<td>'H'</td>
</tr>
<tr>
<td>'H'</td>
</tr>
<tr>
<td>''</td>
</tr>
<tr>
<td>''</td>
</tr>
</tbody>
</table>
Python has single-quoted and double-quoted strings:

<table>
<thead>
<tr>
<th>&quot;Hello, World!&quot;</th>
<th>'Hello, World!'</th>
</tr>
</thead>
<tbody>
<tr>
<td>'Hello, World!'</td>
<td>'Hello, World!'</td>
</tr>
</tbody>
</table>

Whether we use single or double quotes, the result is the same. However, having both is convenient:

<table>
<thead>
<tr>
<th>&quot;Let's play!&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Let's play!&quot;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>'I said, &quot;We want to play!&quot;'</th>
</tr>
</thead>
<tbody>
<tr>
<td>'I said, &quot;We want to play!&quot;'</td>
</tr>
</tbody>
</table>
What if we have both single and double quotes (or other special characters) in a string? We can escape them using the backslash character:

'I said, "Let\'s play!"

Alternatively, we can wrap the string in triple quotes:

"""Single ' and double " quote."""

'Single \' and double " quote.'
Triple quotes also allow us to create multi-line strings:

"""This is
a multiline string."""

'This is\na multiline string.'

If we want to write a single-line string that is too long to put on one line in our code, we can use the backslash character:

'The linebreak \is ignored.'

'The linebreak is ignored.'
An alternative way to do that is via string concatenation:

```
part_1 = 'This is a '
part_2 = 'single-line string.'
part_1 + part_2

'This is a single-line string.'
```
A string is a sequence of characters, and we can access individual characters by indexing:

<table>
<thead>
<tr>
<th>'Hello, World!'[0]</th>
<th>'H'</th>
</tr>
</thead>
<tbody>
<tr>
<td>'Hello, World!'[-1]</td>
<td>'!'</td>
</tr>
</tbody>
</table>

We can get length of a string by using function `len()`:

```
len('Hello, World!')
```

13
Strings are examples of sequences. Other basic sequence types in Python are lists, tuples, and ranges.

<table>
<thead>
<tr>
<th>[1, 2, 3, 4, 5]</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1, 2, 3, 4, 5]</td>
</tr>
<tr>
<td>(1, 2, 3, 4, 5)</td>
</tr>
<tr>
<td>(1, 2, 3, 4, 5)</td>
</tr>
<tr>
<td><code>range(1, 6)</code></td>
</tr>
<tr>
<td><code>range(1, 6)</code></td>
</tr>
</tbody>
</table>
A Python list is a mutable, heterogeneous, ordered sequence of elements:

```python
my_list = [True, 2, 'three']
my_list[1]
2
my_list[2] = 'third'
my_list
[True, 2, 'third']
len(my_list)
3
```
We can add elements to a list via `extend()` and `append()`:

```python
my_list.append('IV')
my_list
[True, 2, 'third', 'IV']

my_list.extend(['101', '110'])
my_list
[True, 2, 'third', 'IV', '101', '110']

my_list.append(7.0)
my_list
[True, 2, 'third', 'IV', '101', '110', 7.0]
```
We can also concatenate lists:

```python
list_1 = [1, 2]
list_2 = [3, 4]
list_3 = [5, 6]
big_list = list_1 + list_2 + list_3
big_list

[1, 2, 3, 4, 5, 6]
```
A Python tuple is an **immutable**, heterogeneous, ordered sequence of elements:

```python
my_tuple = (True, 2, 'three')
my_tuple[1] + len(my_tuple)
```

5

```python
my_tuple[2] = 'third'
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```
We can’t add elements to a tuple (it is immutable). We can still concatenate tuples:

tuple_1 = (1, 2)
tuple_2 = (3, 4)
tuple_3 = (5, 6)
tuple_4 = (7, )
tuple_5 = ()
big_tuple = tuple_1 + tuple_2 + tuple_3 +
    tuple_4 + tuple_5
big_tuple

(1, 2, 3, 4, 5, 6, 7)
Unlike Java, Python doesn’t have a three-statement loop such as

```java
for (int i = 0; i < 5; i++) {
    System.out.println(i);
}
```

*for* loop in Python iterates over elements of a collection, such as a list or a tuple:

```python
for el in [True, 2, 'three']:
    print(el)
```

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>2</td>
<td>three</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Data Types: range**

`range` is another basic sequence type in Python. It is a sequence of integers from start to end:

```python
for el in range(3):
    print(el)
0
1
2
```

```python
for el in range(3, 6):
    print(el)
3
4
5
```

```python
for el in range(0, -10, -4):
    print(el)
0
-4
-8
```
range data type comes in handy when we want to perform iteration a specific number of times or want to iterate through specific numbers. We can do the following:

```python
for el in (0, 1, 2, 3, 4, 5, 6, 7, 8, 9):
    do something
```

Or we can do the following:

```python
for el in range(10):
    do something
```
**CONTROL FLOW: LOOPS**

break statement terminates a loop, continue statement skips to the next iteration of a loop:

```python
for x in range(5, 15):
    if x > 10:
        break
    if x % 2 == 0:
        continue
    print(x)
```

5
7
9
In Java, a block of code is delimited by curly braces `{}`. In Python, a block of code is marked by indentation. Indent your blocks of code with four spaces!
A Python set is a mutable, heterogeneous, unordered sequence of distinct elements. A set can contain only hashable elements (for now, read this as immutable).

```python
define my_set = {'string', 4, 2, 1, 2, 3, 'string'}
define my_set
\{1, 2, 3, 4, 'string'\}
```
### Data Types: Set

We can add elements to a set using the `add()` method:

```
my_set.add('string2')
my_set
{1, 2, 3, 4, 'string2', 'string'}
```

Membership operators `in` and `not in` allow us to check whether an element is present in a set:

```
'string2' in my_set
True

'string3' not in my_set
True
```
We can compute union, intersection, difference, and symmetric difference of sets in Python:

<table>
<thead>
<tr>
<th>my_set_1</th>
<th>my_set_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>{1, 2, 3, 4, 5}</td>
<td>{4, 5, 6, 7, 8}</td>
</tr>
</tbody>
</table>

my_set_1 | my_set_2
---------|---------
{1, 2, 3, 4, 5, 6, 7, 8}

my_set_1 & my_set_2
---------|---------
{4, 5}

my_set_1 - my_set_2
---------|---------
{1, 2, 3}

my_set_1 ^ my_set_2
---------|---------
{1, 2, 3, 6, 7, 8}
A dictionary is a mapping from hashable values to arbitrary values. Another way to describe it is as a mutable, heterogeneous, unordered sequence of key-value pairs:

```python
define my_dict = {'c': 1, 'b': 2, 'a': 3}
define my_dict['d'] = 4
my_dict['c'] = 'Hello, World!'
my_dict

{'c': 'Hello, World!', 'b': 2, 'a': 3, 'd': 4}
```

```python
my_dict['b']

2
```
Can we concatenate two dictionaries the way we concatenated tuples and lists? No:

```
my_dict + {'z': True}
```

Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'dict' and 'dict'
We can, however, merge two dictionaries using the `update()` method:

```python
my_dict1 = {'a': 1, 'b': 2}
my_dict2 = {'a': 3, 'c': 4}
my_dict1.update(my_dict2)
my_dict1
```

```
{'a': 3, 'b': 2, 'c': 4}
```
### Data Types: Conversion

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
<th>Expected Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bool(10)</code></td>
<td></td>
<td>True</td>
</tr>
<tr>
<td><code>bool(0)</code></td>
<td></td>
<td>False</td>
</tr>
<tr>
<td><code>bool('string')</code></td>
<td></td>
<td>True</td>
</tr>
<tr>
<td><code>bool('')</code></td>
<td></td>
<td>False</td>
</tr>
<tr>
<td><code>bool([1])</code></td>
<td></td>
<td>True</td>
</tr>
<tr>
<td><code>bool([])</code></td>
<td></td>
<td>False</td>
</tr>
</tbody>
</table>
### Data Types: Conversion

<table>
<thead>
<tr>
<th>int(True)</th>
<th>int(5.9)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>int(False)</th>
<th>int('10')</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>int(5.4)</th>
<th>int('10', 2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>
## Data Types: Conversion

<table>
<thead>
<tr>
<th>float(True)</th>
<th>float(5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>5.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>float(False)</th>
<th>float('10.0')</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>10.0</td>
</tr>
<tr>
<td>str(True)</td>
<td>str(12.5)</td>
</tr>
<tr>
<td>--------------</td>
<td>----------------</td>
</tr>
<tr>
<td>'True'</td>
<td>'12.5'</td>
</tr>
<tr>
<td>str(12)</td>
<td>str([1, 2, 3])</td>
</tr>
<tr>
<td>'12'</td>
<td>'12, 2, 3]'</td>
</tr>
</tbody>
</table>
## Data Types: Conversion

<table>
<thead>
<tr>
<th>Python Type</th>
<th>Equivalent Type</th>
<th>Example</th>
<th>Equivalent Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>list((1, 2, 3))</code></td>
<td>List</td>
<td><code>[1, 2, 3]</code></td>
<td><code>list([1, 2, 3])</code></td>
</tr>
<tr>
<td><code>tuple([1, 2, 3])</code></td>
<td>Tuple</td>
<td><code>(1, 2, 3)</code></td>
<td><code>tuple((1, 2, 3))</code></td>
</tr>
<tr>
<td><code>list({'a': 1, 'b': 2})</code></td>
<td>List of dictionary items</td>
<td><code>['a', 'b']</code></td>
<td><code>list([{'a': 1, 'b': 2}])</code></td>
</tr>
<tr>
<td><code>tuple({'a': 1, 'b': 2})</code></td>
<td>Tuple of dictionary items</td>
<td><code>('a', 'b')</code></td>
<td><code>tuple([{'a': 1, 'b': 2}])</code></td>
</tr>
</tbody>
</table>
### Data Types: Conversion

<table>
<thead>
<tr>
<th>Code</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>set([1, 2, 3, 1])</code></td>
<td><code>{1, 2, 3}</code></td>
</tr>
<tr>
<td><code>set((1, 2, 3))</code></td>
<td><code>{1, 2, 3}</code></td>
</tr>
<tr>
<td><code>set({'a': 1, 'b': 2})</code></td>
<td><code>{'a', 'b'}</code></td>
</tr>
</tbody>
</table>
Python: Data Types

- int
- float
- bool
- None

- str
- list
- tuple
- range

- set
- range

Type conversion
Python: Control Flow

- if
- if else
- if elif
- else
- while
- for in
A function is a named block of code that can accept arguments and can have a return value. We have already seen a function that returns a number of elements in a sequence:

```
len('Hello, World!')
```

13

How can we define our own functions?
To define a function, we use the keyword ‘def‘ followed by a function name, a list of parameters in parentheses, a colon, and the function body:

```python
def print_square(x):
    print(x * x)
```

```
print_square(9)
```

```
81
```

We use the same indentation with four spaces to delimit a function’s body.
Functions can have an arbitrary number of parameters:

```python
def really_stale_joke():
    print(42)

really_stale_joke()
42

def my_sum(first, second, third, fourth):
    print(first + second + third + fourth)

my_sum(10, 9, 8, 7)
34
```
Functions can have a return value:

```python
def my_pow(x, power):
    return x ** power

cube_of_four = my_pow(4, 3)
cube_of_four
```

```
64
```
pass statement allows us to create a function with an empty body:

```python
def empty_func():
    pass

empty_func()
```
Use lowercase for the variable names, with words separated by underscores (just like the variables).
The class mechanism allows us to create new data types, combining state and behavior. To create a new class, use keyword `class`:

```python
class Duck:
    pass

my_duck = Duck()
type(my_duck)

<class '__main__.Duck'>
```
To initialize an object’s state, we use the `__init__` method. This method is automatically invoked after a new object has been instantiated:

```python
class Duck:
    def __init__(self, name, color):
        self.name = name
        self.color = color

my_duck = Duck('Pinky', 'pink')
print(my_duck.name)
print(my_duck.color)
```

Pinky
pink
When we invoke a method on an object, the object is automatically passed to the method as the first parameter. We can give the first parameter any name we want, but conventionally it is called ‘self’:

```python
class UnusualDuck:
    def __init__(this, name, color):
        this.name = name
        this.color = color

my_duck_2 = UnusualDuck('Greeny', 'green')
print(my_duck_2.color)
print(my_duck_2.name)
```

green
Greeny
We define methods just like we define functions, except we do it inside a class:

```python
class Duck:
    def __init__(self, name, color):
        self.name = name
        self.color = color
    def talk(self):
        return "Quack! I'm " + self.name

my_duck = Duck('Pinky', 'pink')
my_duck.talk()
"Quack! I'm Pinky"
```
We can create a subclass by specifying its superclass in a ‘class’ statement. Let’s define the classes:

class Animal:
    def __init__(self, name, color):
        self.name = name
        self.color = color

class Duck(Animal):
    def talk(self):
        return "Quack! I'm " + self.name

class Cow(Animal):
    def talk(self):
        return "Moo! I'm " + self.name
Now, let’s instantiate the classes:

```python
my_duck = Duck('Pinky', 'pink')
my_duck.talk()
"Quack! I'm Pinky"

my_cow = Cow('Browny', 'brown')
my_cow.talk()
"Moo! I'm Browny"
```
We can override a method of a superclass by reimplementing it in a subclass:

class Animal:
    def __init__(self, name, color):
        self.name = name
        self.color = color
    def talk(self):
        return "Hi! I'm " + self.name

class Cow(Animal):
    def talk(self):
        return "Moo! I'm " + self.name

Cow('Browny', 'brown').talk()

"Moo! I'm Browny"
We can invoke a method of a superclass via ‘super()‘ function:

class ChocoCow(Cow):
    def __init__(self, name, color, cocoa_content):
        super().__init__(name, color)
        self.cocoa_content = cocoa_content

choco_cow = ChocoCow('Goldy', 'golden', 65)
choco_cow.talk()

"Moo! I'm Goldy"

choco_cow.cocoa_content

65
Class names should follow upper camel case convention (WriteYourClassNamesLikeThis). The same restrictions on characters apply.
We can develop code in Python. We can make it persistent by saving a Jupyter Notebook. How do we save our code as a program that can be run later? Well, we can save it in a file with a `.py` extension and run it via `python filename.py`, but what if we want to develop something more complex than a script? We need modules.
Let’s say we decided to develop a bunch of functions implementing matrix multiplication and function differentiation. How can we group them together for easy access in the future, like a... library? We need modules.
We decided to develop a computer game about charming rogues, palace intrigues, and time travel that has the following characters:

- Robin Hood, who likes to shoot arrows with a **bow**;
- Antonio Vivaldi, who likes to play on a violin with a **bow**;
- Marie Antoinette, who likes to tie her hair with a **bow**;
- ...

You get the idea. We have a naming conflict. We need modules.
Python modules give us a mechanism to solve problems of code persistency, library organisation, and naming conflicts.
A Python module is a file containing Python code and having `.py` extension. The name of the module is the name of the file without extension. Let us create two such files:

Listing 1: greetings_en.py
```python
def hello_world():
    return 'Hello, World!'
```

Listing 2: greetings_de.py
```python
def hello_world():
    return 'Hallo, Welt!'
```
Now, we can load our modules by name using `import`:

```python
import greetings_en
import greetings_de
greetings_en.hello_world()
'Hello, World!'

greetings_de.hello_world()
'Hallo, Welt!'```
We can specify a custom name for a module using `import as`. We can also import specific names from a module:

```python
import greetings_de as dtsch
dtsch.hello_world()

'Hallo, Welt!'
```

```python
from greetings_en import hello_world
hello_world()

'Hello, World!'
```

```python
from greetings_en import hello_world as oi
oi()

'Hello, World!'
```
Python comes with many built-in modules, e.g., datetime:

```python
from datetime import datetime

time_start = datetime.now()

print(time_start)

datetime.datetime(2020, 4, 11, 15, 59, 58, 436321)
```

We can install additional modules with conda via ‘install‘ command:

```bash
conda install biopython
```
This ends our shallow and narrow overview of Python. Hopefully you all now have a basic understanding of the following topics:

- Modules
- Classes
- Functions
- Control Structures
- Basic Data Types
- Variables
- PEP 8 – Style Guide for Python Code
  - https://www.python.org/dev/peps/pep-0008/
- The Python Tutorial
  - https://docs.python.org/3/tutorial/
- The Python Language Reference
  - https://docs.python.org/3/reference/
- The Python Standard Library Documentation
  - https://docs.python.org/3/library/
- ‘help‘ and ‘dir‘ functions :-)