Lecture 2: Python programming

Building blocks and tools

Nils Olovsson
nils.olofsson@igp.uu.se

Introduction to data science and Python programming for medical research
Uppsala University

2025

UPPSALA
UNIVERSITET

Content

: Recap

- Types, integers, floats, strings and bool.
- Mathematical-like operators variables, `+`, `*` etc.
- Comparisons and Boolean values.
- If-elif-else statements.
- Installing softwares.

Exceptions

Exceptions

Exceptions: `try`

There are many operations that are **not allowed**. Attempting such operations can raise an **exception**.

We can but such sensitive code inside a **try** block. If there is an exception it can be handled in the following **except** block.

(Note that these are **not errors** as long as we can handle them.)

```
a = 3
print(f"Value of a before: {a}")
try:
    # Dividing by zero is not allowed and will
    # raise an exception.
    #
    # Since in the try-block, the program
    # will not crash but will instead go
    # to the except-block where we can
    # attempt to handle it.
    b = 3 / 0
    a = 5
except Exception as e:
    print("!!! Caught an exception !!!")
    print(e)
    pass
# Value stays the same since code after zero
# division in the try-block was never executed
print(f"Value of a after: {a}")
```

```
>> Value of a before: 3
>> !!! Caught an exception !!!
>> division by zero
>> Value of a after: 3
```

https://docs.python.org/3/library/exceptions.html

Containers

Containers

Containers: List

A list is a sequential, dynamic, container.

Created with brackets `l = []` or with `list()`.

Can store any types, integers, floats, strings, etc.

Even other lists (called **nested list**)!

A list does not have to only contain one type, they can be mixed.

Elements can be accessed by **indexing** or by **slicing**.

```
# Initialize a list of integers
a = [9, 3, 6, 2, 1]
# Initialize a nested list of mixed types
b = [1, [6, 'hi'], [9, 2]]

# Print the two lists and inspect the type
print(f"a ({type(a)}): {a}")
print(f"b ({type(b)}): {b}")

# Use index to get a value (Begins at 0)
print(a[0])
# Use index to set a value in the list
print(a)
a[3] = 8
print(a)
# Slicing with "[start:end]"to get a sub-list
print(a[1:3])
```

```
>> a (<class 'list'>): [9, 3, 6, 2, 1]
>> b (<class 'list'>): [1, [6, 'hi'], [9, 2]]
>> 9
>> [9, 3, 6, 2, 1]
>> [9, 3, 6, 8, 1]
>> [3, 6]
```

Containers: List growing

Elements **can be added** to a list using `append()`.

The size can be inspected using the Python `len()` command.

Lists can be joined with the + operator. Results in a new list.

List methods

- append
- clear
- copy
- count
- extend
- index
- insert
- pop
- remove
- reverse
- sort

```python
# Initialize an empty list
a = []
print(a)

# Use append() to grow the list
a.append(2)
a.append(5)
a.append(57)
print(a)

# Inspect the size of the list with len()
n = len(a)
print(f"Size of list: {n}")

# Append a second list to the first
b = [1, 2, 3]
c = a + b
print(b)
print(c)
```

```
>> []
>> [2, 5, 57]
>> Size of list: 3
>> [1, 2, 3]
>> [2, 5, 57, 1, 2, 3]
```

Containers: Mutable

In a list we can swap out items and we append new items.
A list is **mutable**, it can be altered.

A string is also a form of container that can be indexed and sliced just as a list. But we are not allowed to change characters in the string or appending to it.
Strings are **immutable**.

If we want to alter a string, we have to create a new string.

```
s = 'Hello'
l = list(s)

# Lists are mutable so we can swap elements
print(f"Before: {l}")
l[0] = 'b'
print(f"After:  {l}")

# Can't assign a new value to the first
# string element! Strings are immutable!
try:
    s[0] = 'b'
except Exception as e:
    print(e)
# We can however create a new string
s0 = 'b' + s[1:] # By concat. sliced string
s1 = ''.join(l)  # From list using join
print(s0)
print(s1)
```

```
>> Before: ['H', 'e', 'l', 'l', 'o']
>> After:  ['b', 'e', 'l', 'l', 'o']
>> 'str' object does not support item
     assignment
>> bello
>> bello
```

Containers: Tuples

A tuple is a sequential, **immutable**, container. Created with parenthesis `l = ()` or with `tuple()`.

Can store any types, integers, floats, strings, lists, etc.

Even other tuples (called **nested**)!

A tuple does not have to only contain one type, they can be mixed.

Elements can be retrieved by **indexing** or by **slicing**.

The tuple can't be altered once created!

```
# Initialize a tuple of integers
a = (9, 3, 6, 2, 1)
# Initialize a nested tuple of mixed types
b = (1, (6, 'hi'), [9, 2])

# Print the two tuples and inspect the type
print(f"a ({type(a)}): {a}")
print(f"b ({type(b)}): {b}")

# Use index to get a value (Begins at 0)
print(a[0])
# Slicing with "[start:end]"to get a sub-tuple
print(a[1:3])
# We can't index to change an element.
# Since tuples are immutable!
try:
    a[3] = 8
except Exception as e:
    print(e)
```

```
>> a (<class 'tuple'>): (9, 3, 6, 2, 1)
>> b (<class 'tuple'>): (1, (6, 'hi'), [9, 2])
>> 9
>> (3, 6)
>> 'tuple' object does not support item
     assignment
```

Containers: Dictionaries

A dictionary is a **non-sequential**, mutable, container. Created with curly brackets `d = {}` or with `dict()`.

Consists of pairs of **keys** and **values**. Elements are retrieved by **key**. Dictionaries can't be sliced.

Values can be any type.
Keys can be any **immutable** type.

Dictionary methods

- clear
- copy
- fromkeys
- get
- items
- keys
- pop
- popitem
- setdefault
- update
- values

```
1  # Initialize empty dictionary and add elements
2  d = {}
3  d['name'] = 'Valeria'
4  d['age'] = 1.5
5  d['occupation'] = 'pre-school'
6  print(type(d))
7  print(d)
8
9  # Initialize filled dictionary
10 # Key-value pairs are separated by ":"
11 d = {'name': 'Squeek',
12       'age': 12,
13       'occupation': 'dog'}
14 print(d)
15
16 # Can access elements using the keys.
17 print(d['name'])
18 print(d['age'])
```

```
1  >> <class 'dict'>
2  >> {'name': 'Valeria', 'age': 1.5, 'occupation
       ': 'pre-school'}
3  >> {'name': 'Squeek', 'age': 12, 'occupation':
        'dog'}
4  >> Squeek
5  >> 12
```

Containers: Dictionary elements

Using a key that is not present in the dictionary will cause and error.
Need to **check before retrieving** if we are not sure!

Inserting using existing key will overwrite previous value.

Lists, tuples and strings can all be concatenated using the + operator.
Dictionaries can't be concatenated.
They can however be **merged**.

```
# Initialize dictionaries
d0 = {'name': 'Valeria', 'age': 1.5}
d1 = {'weight': 12.4}
print(d0)
print(d1)
# Check if key in dictionary using keys()
if 'name' in d0.keys():
    print(d0['name'])
# Shorthand check if key is in dictionary
if 'name' in d0:
    print(d0['name'])

if 'occupation' in d0:
    print(d0['occupation'])
else:
    print("Key 'occupation' not in d0")

# Merge with union set operator
m = d0 | d1
print(m)
```

```
>> {'name': 'Valeria', 'age': 1.5}
>> {'weight': 12.4}
>> Valeria
>> Valeria
>> Key 'occupation' not in d0
>> {'name': 'Valeria', 'age': 1.5, 'weight':
     12.4}
```

Containers: String

Strings are a form of sequential, **immutable**, container.
They can only contain **characters**.

Can be indexed and sliced to access.
But can not be written to.

Strings are associated with many built in methods, of which a few are shown in this list.

String methods

- capitalize
- casefold
- center
- count
- encode
- endswith
- expandtabs
- find
- format
- format_map
- index
- isalnum
- isalpha
- isascii
- ...

```
1  # Initialize a string
2  s = 'hello'
3  print(s)
4
5  # Strings can be sliced
6  # and concatenated. Create a
7  # new string and assign to old variable
8  s = s[0:2] + 'y there!'
9  print(s)
10
11 # Lets try one of the built in methods
12 s = s.capitalize()
13 print(s)
```

```
1 >> hello
2 >> hey there!
3 >> Hey there!
```

Control flow

Control flow: `for` loop

Previously saw how program flow can be controlled by executing **branches** using conditional `if` blocks.

The other control of flow is the **repetition** of a block of code.
E.g. to perform the same operation on each row of spreadsheet data.

Achieved by **for** and **while loops**.

A for loop can be used to **iterate over** a sequence of **numbers** or the **elements in** a sequential container, *e.g.* a **list**.

```python
# Initialize a list of values
l = [1, 2, 3, 4, 5]

# Compute the sum of all values in the list
s = 0 # Initialize sum
for v in l:
    print(f"Value: {v}")
    s += v

# Print original list and computed sum
print(f"Values in list: {l}")
print(f"Sum of values: {s}")
```

```
>> Value: 1
>> Value: 2
>> Value: 3
>> Value: 4
>> Value: 5
>> Values in list: [1, 2, 3, 4, 5]
>> Sum of values: 15
```

Control flow: `while` loop

The **while** loop works similarly to the for loop.

Instead of specifying a set of numbers or elements to iterate through the while block is repeated until some **conditional statement** is `False`.

A while loop is better to use when we **don't know how many repetitions** we will perform.

```
# Initialize a list of values
l = [1, 2, 3, 4, 5]

# Compute the sum of all values in the list
# Equivalent to the 'for' loop seen previously
s = 0      # Initialize sum
index = 0 # Need manual list index
while index < len(l):
    v = l[index]
    print(f"Value: {v}")
    s += v
    # Need to manually increment the index
    index += 1

# Print original list and computed sum
print(f"Values in list: {l}")
print(f"Sum of values: {s}")
```

```
>> Value: 1
>> Value: 2
>> Value: 3
>> Value: 4
>> Value: 5
>> Values in list: [1, 2, 3, 4, 5]
>> Sum of values: 15
```

Control flow: Nesting and indentation

All blocks that control flow can be **nested**, *i.e.* can put *e.g.* an `if` block inside a `for` loop block.

This is done by increasing the **indentation level** to divide the code into blocks.

```python
# Initialize a list of values
l = [1, 2, 3, 4, 5, 6, 7, 8]

# Compute sum of all EVEN values in list!
s = 0 # Initialize sum
# Main (block #1)
for v in l:
    # For (block #2)
    # Check if value is evenly divisible by 2
    # (Remainder is zero)
    if v % 2 == 0:
        # If (block #3)
        print(f"Even value: {v}")
        s += v
    # For (block #2)
# Main (block #1)

# Print original list and computed sum
print(f"Values in list: {l}")
print(f"Sum of even values: {s}")
```

```
>> Even value: 2
>> Even value: 4
>> Even value: 6
>> Even value: 8
>> Values in list: [1, 2, 3, 4, 5, 6, 7, 8]
>> Sum of even values: 20
```

Control flow: Break and continue

`continue`

Parts of the block in a loop can be skipped using the **continue** statement. This will cause the loop to **jump to the next iteration**.

`break`

Loops can be **exited** using a **break** statement.

Using these can decrease the readability of the code but are perfectly valid.

```python
# Initialize a list of values
l = [1, 2, 3, 4, 5, 6, 7, 8]

# Compute sum for only odd elements but only
# until the sum has grown larger than 5
s = 0 # Initialize sum
count = 0
for v in l:
    # Skip even values
    if v % 2 == 0:
        continue
    print(f"Odd value: {v}")
    s += v
    count += 1
    # Break is sum is larger than 5
    if s > 5:
        break

# Print original list and computed sum
print(f"Values in list: {l}")
print(f"Sum of first {count} odd values: {s}")
```

```
>> Odd value: 1
>> Odd value: 3
>> Odd value: 5
>> Values in list: [1, 2, 3, 4, 5, 6, 7, 8]
>> Sum of first 3 odd values: 9
```

Control flow: Enumerate and range

enumerate

In **for** it is often desirable to get both the **value** and the **index**.
Is done by enclosing the container, *e.g.* the list, in an **enumerate** statement.

```python
# Initialize a list of values
l = [1, 2, 3, 4, 5]

# Print index and value of all elements
for index, v in enumerate(l):
    print(f"{index}: {v}")

# Compute sum of numbers 1 to 6
# Not including 6! Range = [start, stop)
s = 0 # Initialize sum
for i in range(1, 6):
    s += i
# Print original list and computed sum
print(f"Sum of numbers in range(1, 6): {s}")
```

range

It is also very common that we want to run the loop for a **fixed number of times**.
This can be done by iterating over a **range** of numbers.

```
>> 0: 1
>> 1: 2
>> 2: 3
>> 3: 4
>> 4: 5
>> Sum of numbers in range(1, 6): 15
```

Control flow: Iterate over dictionaries

Dictionaries are **not sequential** but we can **iterate over the keys or** by getting the **item pairs** using `items()`.

```
# Initialize empty dictionary and add elements
d = {}
d['name'] = 'Valeria'
d['age'] = 1.5
d['occupation'] = 'pre-school'

# Iterate over all keys
print("Iterate over key-value pairs.")
for key in d.keys():
    # Need to manually retrieve value
    value = d[key]
    print(f"{key}: {value}")

# Iterate over all key-value pairs using the
# more compact items()
print("Iterate over key-value pairs (again)")
for key, value in d.items():
    print(f"{key}: {value}")
```

```
>> Iterate over key-value pairs.
>> name: Valeria
>> age: 1.5
>> occupation: pre-school
>> Iterate over key-value pairs (again)
>> name: Valeria
>> age: 1.5
>> occupation: pre-school
```

Control flow: List comprehensions

List comprehensions can be used to compress for loops into single line expressions.

(This saves a little space but can make the code harder to read!)

```python
# Initialize a list using a for loop
a = []
for i in range(5):
    if i%2 == 0:
        a.append(f"Number: {i}")

# Initialize an equal list using
# list comprehension.
# This compresses the expression that was
# three rows into a single row!
b = [f"Number: {i}" for i in range(5) if i
    %2==0]

# Print the lists
print("The lists are the same:")
print(f"a: {a}")
print(f"b: {b}")
```

```
>> The lists are the same:
>> a: ['Number: 0', 'Number: 2', 'Number: 4']
>> b: ['Number: 0', 'Number: 2', 'Number: 4']
```

```
[expression for item in container condition]
```

Functions

Functions

Functions: def

As the size of your scripts grow you will want to put parts of it into **separate blocks that can be accessed from elsewhere**.

Place code in **functions**.

Start with def statement, then function name, followed by parentheses that can enclose a number of **input arguments**.

We then **call** the function using an analogous syntax.

Functions may also **return** one or a tuple of results.

```
# Define function that computes and returns
# the sum of two numbers
def compute_sum(a, b):
    s = a + b
    print(f"{a} + {b} = {s}")
    return s

# Call the function to compute sums.
# Assign the results of the function
# evaluations to variables.
s0 = compute_sum(3.0, 5.5)
s1 = compute_sum(1.5, 4.0)
s2 = compute_sum(7.0, 2.0)

# Print the returned sums
print(f"Returned sum: {s0}")
print(f"Returned sum: {s1}")
print(f"Returned sum: {s2}")
```

```
>> 3.0 + 5.5 = 8.5
>> 1.5 + 4.0 = 5.5
>> 7.0 + 2.0 = 9.0
>> Returned sum: 8.5
>> Returned sum: 5.5
>> Returned sum: 9.0
```

Functions: Optional arguments

Optional arguments

A function can have a set of **optional** arguments that **default** to a predefined value if not provided.

The optional arguments must come after the required ones!

Advanced program flow

Before functions our scripts executed in **top-to-bottom** order, with some branching and repetitions.

Now, code execution can **jump** around. From the main block, to a function, to another function, etc.

```python
# Define function that computes and returns
# the sum of three numbers.
# Only two input arguments are required,
# the third one defaults to 0.0.
def compute_trisum(a, b, c=0.0):
    s = a + b + c
    print(f"{a} + {b} + {c} = {s}")
    return s

# Call the function to compute sums.
# Assign the results of the function
# evaluations to variables.
s0 = compute_trisum(2.0, 5.5, 2.5)
s1 = compute_trisum(1.5, 4.0)

# Print the returned sums
print(f"Returned sum: {s0}")
print(f"Returned sum: {s1}")
```

```
>> 2.0 + 5.5 + 2.5 = 10.0
>> 1.5 + 4.0 + 0.0 = 5.5
>> Returned sum: 10.0
>> Returned sum: 5.5
```

Functions: Docstring

Functions should have a descriptive comment at the start of the function block, a **docstring**.

It can be **accessed by the IDE** to provide in-line descriptions of functions.

(Input arguments and returned results can also be specified. But a better way is to use **type hints**)

```python
# Define function that computes and returns
# the sum of two numbers.
# This is the same function as before.
def compute_sum(a, b):
    """ Compute the sum of two numbers.
    This is a docstring that describes the
    function. It calculates the sum of two
    numbers using the '+' operator.
    """
    s = a + b
    print(f"{a} + {b} = {s}")
    return s

# Call the function to compute sums.
s0 = compute_sum(3.0, 5.5)
s1 = compute_sum(1.5, 4.0)

# Print the returned sums
print(f"Returned sum: {s0}")
print(f"Returned sum: {s1}")
```

```
>> 3.0 + 5.5 = 8.5
>> 1.5 + 4.0 = 5.5
>> Returned sum: 8.5
>> Returned sum: 5.5
```

Functions: Type hints

Python is a **dynamically typed language**, *i.e.* we **don't care about the types** of variables.

However, it facilitates both the **automatic type deduction** in IDEs and human readability to add explicit **type hints.**

```python
# This is the same function as before.
# Type hints have been added for both
# arguments and the return value
def compute_sum(
    a: float,
    b: float,
) -> float:
    """ Compute the sum of two numbers.
    This is a fully documented and
    type hinted function!
    """
    s = a + b
    print(f"{a} + {b} = {s}")
    return s

# Call the function to compute sums.
s0 = compute_sum(3.0, 5.5)
s1 = compute_sum(1.5, 4.0)

# Print the returned sums
print(f"Returned sum: {s0}")
print(f"Returned sum: {s1}")
```

```
>> 3.0 + 5.5 = 8.5
>> 1.5 + 4.0 = 5.5
>> Returned sum: 8.5
>> Returned sum: 5.5
```

Modules

Modules: Structure in separate files

We can go further and put our **function** in a **separate file** from our main script.

We then **import** that file and the function as a **module**.

Math.py

```
1  def compute_sum(
2      a: float,
3      b: float,
4  ) -> float:
5      """ Compute the sum of two numbers.
6      This is a fully documented and
7      type hinted function!
8      """
9      s = a + b
10     print(f"{a} + {b} = {s}")
11     return s
```

Main script

```
1  import Math
2
3  # Call the function from the imported
4  # module to compute sums.
5  s0 = Math.compute_sum(3.0, 5.5)
6  s1 = Math.compute_sum(1.5, 4.0)
7
8  # Print the returned sums
9  print(f"Returned sum: {s0}")
10 print(f"Returned sum: {s1}")
```

```
1  >> 3.0 + 5.5 = 8.5
2  >> 1.5 + 4.0 = 5.5
3  >> Returned sum: 8.5
4  >> Returned sum: 5.5
```

Modules: Import

Can import all contents in the module or a subset.

- `import`
- `import as name`
- `import.subset`
- `import *`

Can import it using a shorthand name. There are several such shorthand names that are standard for the packages we will use in this course.

```
# Import using a shorthand name 'sm'
import Math as sm

# Call the function from the imported
# module to compute sums.
s0 = sm.compute_sum(3.0, 5.5)
s1 = sm.compute_sum(1.5, 4.0)

# Print the returned sums
print(f"Returned sum: {s0}")
print(f"Returned sum: {s1}")
```

```
>> 3.0 + 5.5 = 8.5
>> 1.5 + 4.0 = 5.5
>> Returned sum: 8.5
>> Returned sum: 5.5
```

Standard library

Standard library: Utilities

The **python standard library** contains many **modules** with common utilities. These are four examples that I commonly use.

- `math`
- `random`
- `os`
- `sys`

```python
import sys
import math

# Get platform information
s = sys.platform
print(f"Platform: {s}")

# Compute cosine of 45 degrees
v = math.cos(math.pi * 0.25)
print(v)
```

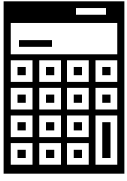
```
>> Platform: linux
>> 0.7071067811865476
```

https://docs.python.org/3/library/index.html

Standard library: `math`

The **math** library contains many **mathematical functions** and some **mathematical constants**.

(We will later look at the **numpy** package. A **specialized mathematics package** for arrays which makes it more suitable for data processing.)



https://docs.python.org/3/library/math.html

```python
import math

# Mathematical constants
pi = math.pi
e = math.e
print(f"pi = {math.pi:.4f}")
print(f"e = {math.e:.4f}")

# Trigonometric function and inverse
freq_mod = 0.3
v = math.cos(freq_mod * math.pi)
a = math.acos(v) / math.pi
print(f"v: {v}")
print("Inverse function restores value")
print(f"{a} = {freq_mod}")
```

```
>> pi = 3.1416
>> e = 2.7183
>> v: 0.5877852522924731
>> Inverse function restores value
>> 0.3 = 0.3
```

Standard library: `random`

The **math** library contains many **mathematical functions** and some **mathematical constants**.

(We will later look at the **numpy** package. A **specialized mathematics package** for arrays which makes it more suitable for data processing.)

```python
import random

# Random float in range [0, 1)
v = random.random()

# Random integer in range [5, 10)
i = random.randint(5, 10)

# Random value from the list
c = random.choice([3, 7, 9])

# Random values
print(f"v = {v:.2f}")
print(f"i = {i}")
print(f"c = {c}")
```

```
>> v = 0.71
>> i = 8
>> c = 3
```

https://docs.python.org/3/library/random.html

Standard library: os

The **os** module is useful for navigating the filesystem.

Can be used to find files, create directories and join path names.

https://docs.python.org/3/library/os.html

```python
import os

# Find all files in this directory
# and iterate through all entries
directory = '.' # This directory
for entry in os.listdir(directory):
    fullpath = os.path.join(directory, entry)

    # Get the file extension and check
    # if entry is a file.
    _, ext = os.path.splitext(entry)
    is_file = os.path.isfile(fullpath)
    is_swp = ext == '.swp'

    # Only print the entry if it is
    # a file but not of type 'swp'
    if is_file and not is_swp:
        print(fullpath)
```

```
>> ./lecture-04-pandas-and-statistics.py
>> ./lecture-05-regression.py
>> ./lecture-06-algebra-matplotlib.py
>> ./lecture-07-clustering.py
>> ./lecture-08-pca.py
>> ./lecture-09-manifold-learning.py
>> ./lecture-10-svm-rf-ai.py
>> ./lecture-13-image-processing.py
>> ./lecture_code_run.py
```

Standard library: sys and the Python path

Python can only find modules if they are on the **Path path**.
These are the directories where the standard library resides and where we typically install packages.

When we write our own modules we should not put them in these directories.

Instead we **add the directories to our path** using **sys**.

```
1  import sys
2
3  # Add a directory to the Python Path
4  directory = os.path.join('..',
5                           'lectures-modules')
6  sys.path.append(directory)
7
8  # Import using a shorthand name 'sm'
9  import Math as sm
10
11 # Call the function from the imported
12 # module to compute sums.
13 s0 = sm.compute_sum(3.0, 5.5)
14 s1 = sm.compute_sum(1.5, 4.0)
15
16 # Print the returned sums
17 print(f"Returned sum: {s0}")
18 print(f"Returned sum: {s1}")
```

```
1  >> 3.0 + 5.5 = 8.5
2  >> 1.5 + 4.0 = 5.5
3  >> Returned sum: 8.5
4  >> Returned sum: 5.5
```

https://docs.python.org/3/library/sys.html

Classes

Classes

Classes: Objects and instances

Python is largely an **object oriented language**.

Objects, like functions, are used to **organize the code**.

Can contain **member variables** and **functions**, sometimes called methods, as **attributes**.
All indented after the `class` declaration.

Should have a constructor `__init__(self, ...)` method.

When you create a **variable of an object** it is called an **instance**.

```python
class Patient:
    def __init__(
        self,
        age: int,
        triage_level: int,
    ):
        """
        The class has two member variables.
        """
        self.age = age
        self.triage_level = triage_level
        return

    def __str__(self) -> str:
        """
        This will automatically be called when
        an instance of the class is cast to
        string, ie in a print statement.
        """
        msg = 'Patient: '
        msg += f'age: {self.age} '
        msg += f'triage: {self.triage_level}'
        msg += ' (ESI)'
        return msg
# Create an instance of the class
p = Patient(76, 2)
print(p)
```

```
>> Patient: age: 76 triage: 2 (ESI)
```

Classes: Attributes and initialization

Should have a **constructor** `__init__(self, ...)` method.

This special function is called when an instance of the class is created.

It is responsible for **initializing** everything inside the class **instance**

Member variables are set.

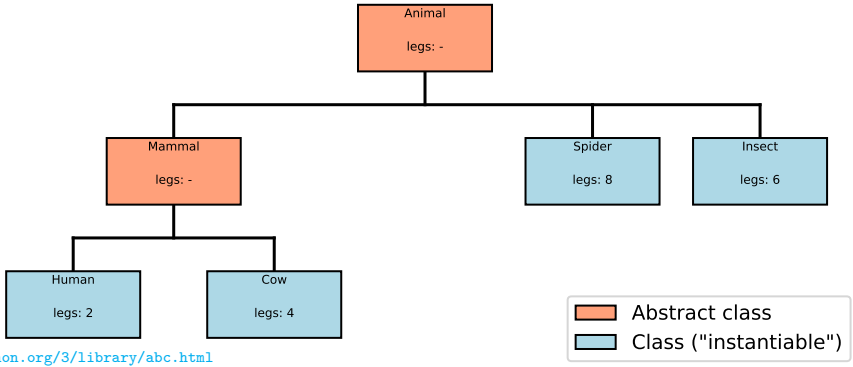
```python
class Patient:
    def __init__(
        self,
        age: int,
        triage_level: int,
    ):
        """
        The class has two member variables.
        """
        self.age = age
        self.triage_level = triage_level
        return

    def __str__(self) -> str:
        """
        This will automatically be called when
        an instance of the class is cast to
        string, ie in a print statement.
        """
        msg = 'Patient: '
        msg += f'age: {self.age} '
        msg += f'triage: {self.triage_level}'
        msg += ' (ESI)'
        return msg
# Create an instance of the class
p = Patient(76, 2)
print(p)
```

```
>> Patient: age: 76 triage: 2 (ESI)
```

Classes: Inheritance (1/2)

A powerful concept in **object oriented programming** is that of **inheritance**. This allows for the **sharing of code** defining **common attributes and methods between different classes**. **Abstract classes** defined common attributes and **can not be instantiated**.



https://docs.python.org/3/library/abc.html

Classes: Inheritance (2/2)

```python
from abc import ABC, abstractmethod
class Animal(ABC):
    def __init__(
        self,
        nr_of_legs: int,
    ):
        self.nr_of_legs = nr_of_legs
        return

    def __str__(self) -> str:
        name = self.__class__.__name__
        msg = f'{name} has '
        msg += f'{self.nr_of_legs} '
        msg += 'legs and says '
        msg += f'"{self.get_sound()}"'
        return msg

    @abstractmethod
    def get_sound(self) -> str:
        pass

class Spider(Animal):
    def __init__(self):
        Animal.__init__(self, 8)
        return

    def get_sound(self) -> str:
        return 'imse vimse'
```

Abstract base classes `ABC` defines classes with **common properties**. Sub-classes then **inherit** from the abstract class.

`@abstractmethod` indicates that **sub-classes** should **define a method with that name and signature**.

```python
class Cow(Animal):
    def __init__(self):
        Animal.__init__(self, 4)
        return

    def get_sound(self) -> str:
        return 'Mooo'

# Create instances of animals
spider = Spider()
cow = Cow()
print(spider)
print(cow)
```

```
>> Spider has 8 legs and says "imse vimse"
>> Cow has 4 legs and says "Mooo"
```

Classes: Public, protected or private

It is common to want to **encapsulate properties of classes** such that they can't be changed from the outside.
Declare such **member variables** or **functions** as **protected** or **private**.

Public `foo`

Accessible from anywhere.

Protected `_foo`

Accessible inside class and any derived classes.

Private `__foo`

Accessible from only the class.

(This is a convention and not enforced by Python!)

```python
class Human:
    def __init__(
        self,
        name: str,
        age: int,
        pn: str,
    ):
        self.name = name
        self._age = age
        self.__pn = pn
        return

    def say_hello(self) -> None:
        msg = ''
        msg += f'I am {self.name}, '
        msg += f'age {self._age} and '
        msg += f'personal number {self.__pn}.'
        print(msg)
        return

human = Human('Nils', 36, '890207')
human.say_hello()
```

```
>> I am Nils, age 36 and personal number
   890207.
```

Classes: Is or has?

```python
class DiseaseAttributes:
    def __init__(
        self,
        r_number: float, mortality: float,
    ):
        self.r_number = r_number
        self.mortality = mortality
        return

class Covid19A(DiseaseAttributes):
    def __init__(
        self,
        r_number: float, mortality: float,
        subtype: str,
    ):
        DiseaseAttributes.__init__(self, r_number, mortality)
        self.subtype = subtype
        return

class Covid19B:
    def __init__(
        self,
        r_number: float, mortality: float,
        subtype: str,
    ):
        self.disease_attributes = DiseaseAttributes(r_number,
     mortality)
        self.subtype = subtype
        return
```

Is covid a **disease or** does it have **disease attributes**?

A design choice whether to use **inheritance or** store properties as **attributes**.

Matter of convenience, **does not have to reflect reality**.

```python
# Create instances
covid19_a = Covid19A(2.5, 0.03, 'delta')
covid19_b = Covid19B(2.5, 0.03, 'delta')
```

Files

Files

Files: Writing and reading

```python
import os
data = [2, 3, 6]
filepath = 'test-file.txt'

# Open and write
with open(filepath, 'w') as f:
    # Write each element to new row
    # by adding a line break '\n'
    for i, v in enumerate(data):
        f.write(f'{i}: {v}\n')
# File is closed after 'with' block.

# Open and read
with open(filepath, 'r') as f:
    # Read all lines
    lines = f.readlines()
    # Print each line
    for line in lines:
        # Line break added by print as default
        # Line break '\n' is already in line
        print(f"line: {line}", end='')
# File is closed after 'with' block.

# Delete our test file
os.remove(filepath)
```

Both **binary** and **text files** can be natively **written** and **read**.

Files are opened using a string signature:

- `'w'` Write
- `'r'` Read
- `'a'` Append

Always open in a `with` block! Files that are not closed will remained locked and behave weird in the OS!

```
>> line: 0: 2
>> line: 1: 3
>> line: 2: 6
```

Files: Pickling

```python
import pickle
filepath = 'test-file.pkl'

# Initialize a nested list and a dict
l = [1, [6, 'hi'], [9, 2]]
d = {'name': 'Nils', 'general-status': 5}

# Put in a tuple and save that to a file using
# pickle. Must open file in byte mode 'b'.
t = (l, d)
with open(filepath, 'wb') as f:
    pickle.dump(t, f)

# Read back the data
t_load = None
with open(filepath, 'rb') as f:
    t_load = pickle.load(f)

print(f"Type of loaded: {type(t_load)}")
for v in t_load:
      print(v)

# Delete our test file
os.remove(filepath)
```

https://docs.python.org/3/library/pickle.html

Any set of Python object instances can be **pickled** into a binary format and **written to file**.

This permits you to **save and load any work and results** you have computed!

```
>> Type of loaded: <class 'tuple'>
>> [1, [6, 'hi'], [9, 2]]
>> {'name': 'Nils', 'general-status': 5}
```

Time

Time

Time: `time` package

Execution time can be measured with the **time** package.

(In IPython time can be measured using `#timeit` magic command.)

```python
import time

t0 = time.time()
for i in range(5):
    print(f"Iteration: {i}")

    # This will pause the execution
    # for 0.1 seconds each iteration
    # in the for loop.
    time.sleep(0.1)
t1 = time.time()

# Compute the difference in time
dt = t1 - t0
print(f"Total time: {dt:.1f} [s]")
```

```
>> Iteration: 0
>> Iteration: 1
>> Iteration: 2
>> Iteration: 3
>> Iteration: 4
>> Total time: 0.5 [s]
```

https://docs.python.org/3/library/time.html

https://ipython.readthedocs.io/en/stable/interactive/magics.html

Analyze and debug

Analyze and debug: Errors and style (`flake8`)

```
1  class BloodPressureTest:
2      def __init__(
3          self,
4          bp_systolic_mmHg,
5          bp_diastolic_mmHg,
6      ):
7          self.bp_systolic_mmHg =
       bp_systolic_mmHg
8          self.bp_diastolic_mmHg =
       bp_diastolic_mmHg
9          return
10
11     def is_elevated(self):
12         if bp_s_mm_Hg>=140.0 or bp_d_mm_Hg
       >=90.0:
13             return True
14         return False
15
16 # Average of three measurements
17 bp_s_mm_Hg = (1/3) * (145+130+150)
18 bp_d_mm_Hg = (1/3) * (95+91+96)
19
20 bp = BloodPressureTest(bp_s_mm_Hg,
21                        bp_d_mm_Hg)
22 print(f"BP is elevated: {bp.is_elevated()}")
```

https://flake8.pycqa.org/en/latest/

Flakes can be used to ensure that the code adheres to **good style**.

```
1 ANALYZEANDDEBUG-flakes-before.py:12:22: E225
      missing whitespace around operator
2 ANALYZEANDDEBUG-flakes-before.py:12:43: E225
      missing whitespace around operator
3 ANALYZEANDDEBUG-flakes-before.py:17:1: E305
      expected 2 blank lines after class or
      function definition, found 1
```

Analyze and debug: Type checks (`mypy`)

Mypy can perform **static type checking** if the code is written using **type hints**.

https://mypy.readthedocs.io/en/stable/

Analyze and debug: Debug (VScode)

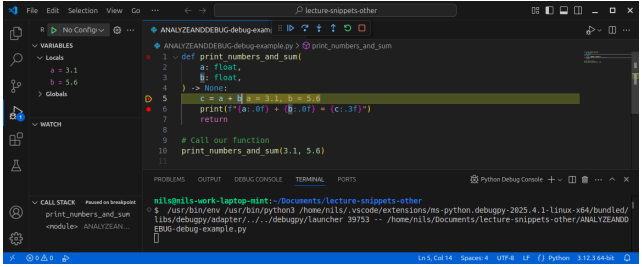
Visual debug in VScode allows you to set **breakpoints**, red dots, at which the execution will pause and values of **variables can be inspected**.

Can you see what is wrong here and why?

```python
def print_numbers_and_sum(
    a: float,
    b: float,
) -> None:
    c = a + b
    print(f"{a:.0f} + {b:.0f} = {c:.1f}")
    return

# Call our function
print_numbers_and_sum(3.1, 5.6)
```

```
>> 3 + 6 = 8.7
```



Analyze and debug: Unit tests (`unittest`)

Unit testing is a popular way of **making sure parts of the code work as expected**.
Especially if it changes often!
Can you follow why one test succeeds and one fail?

```python
import unittest

def multiply(a: float, b: float) -> float:
    return a + b

class TestMultiplication(unittest.TestCase):
    def test_one(self):
        self.assertAlmostEqual(multiply(2.0, 2.0), 4.0)

    def test_two(self):
        self.assertAlmostEqual(multiply(2.0, 6.0), 12.0)

if __name__ == '__main__':
    unittest.main()
```

```
.F
======================================================================
FAIL: test_two (__main__.TestMultiplication.test_two)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/mnt/storage/work-write/courses/datascience-python-course/py/lecture-snippets/lecture-02-python-programming/ANALYZEANDDEBUG-unit-tests.py", line 11, in test_two
    self.assertAlmostEqual(multiply(2.0, 6.0), 12.0)
AssertionError: 8.0 != 12.0 within 7 places (4.0 difference)

----------------------------------------------------------------------
Ran 2 tests in 0.002s

FAILED (failures=1)
```

https://docs.python.org/3/library/unittest.html

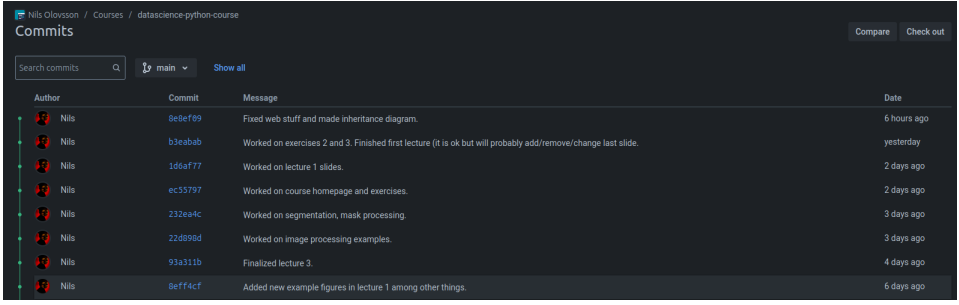
git

git

git: Version control

Version control acts as both a **backup and a development history**. Old version can be **restored** and **compared** to the current or other versions.

Not the only version control system, but it is the **most popular**. **git** itself is **free and open source**.



https://git-scm.com/

git: Hosting

The **git software is free**.

Hosting the code in the cloud **might not be**.

Typically it is *free*. As many things on the internet tend to be.

Technically you can host wherever. Even host yourself.

https://github.com/
https://bitbucket.org
https://about.gitlab.com

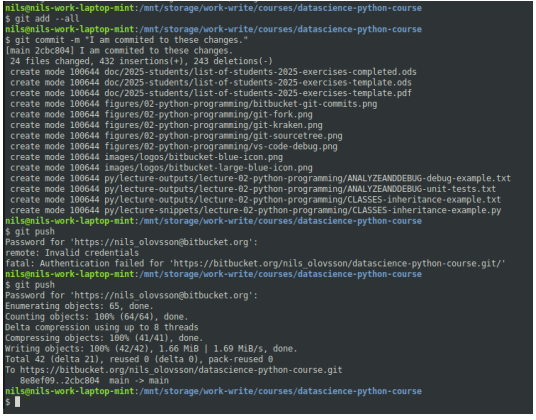
git: Command line

Git is a **command line tool**.

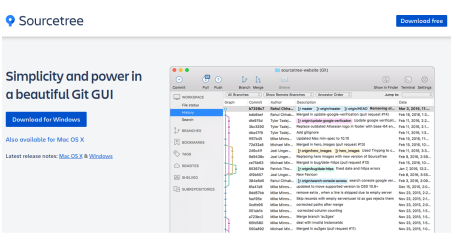
The typical **sequence of commands** to commit the current state of the code is:

1. `git add -all`
2. `git commit -m "A message"`
3. `git push`

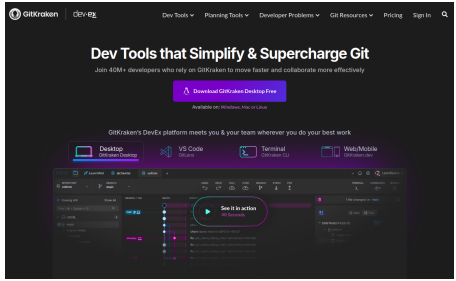
More **advanced features** include the use of **branches**, *e.g.* a development branch and a production branch, and the **resolving of collisions** when more than one person works on the same codebase.



git: Softwares with GUI



A free Git client for Windows and Mac



https://www.sourcetreeapp.com/
https://www.gitkraken.com/
https://git-fork.com/

Packages

Packages

Packages: Installing

(Reminder from yesterday)
Much of what we will use in this course will require **additional Python packages**.
Most packages are also **open source** and they also have several distributors!

If you use **miniconda** there are two package managers you can easily use.

Python package index

```
pip install <PACKAGE_NAME>
```

Conda forge

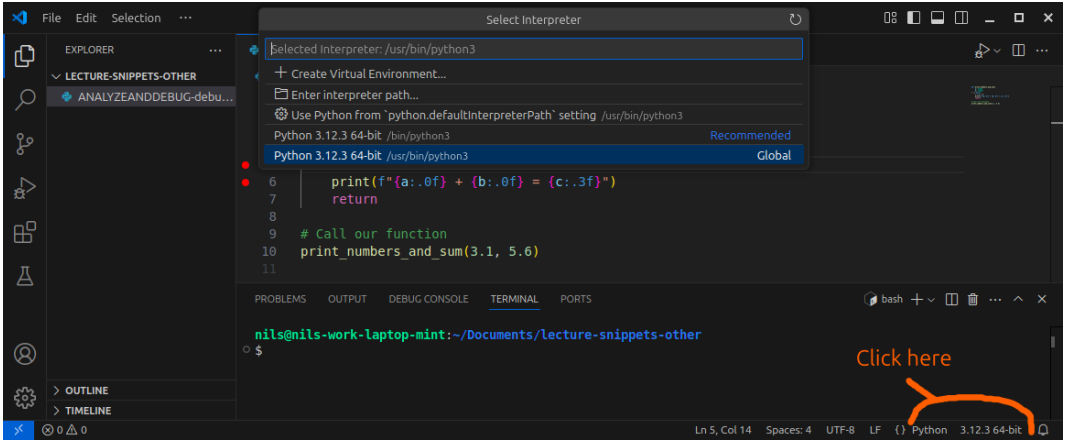
```
conda install <PACKAGE_NAME>
```

https://pypi.org/

https://conda-forge.org/

Packages: Python environment

Once you start using packages you need do be sure you are using the correct Python environment.



: Conclusions

- Exceptions.
- Lists, tuples, dictionaries.
- For and while loops.
- Functions.
- Modules, packages and the standard library.
- Classes and object oriented programming.
- File reading and writing.
- Time measurements.
- Debugging and analyzing.
- Version control using git.