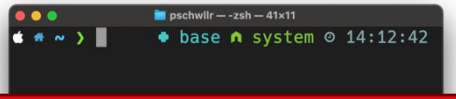
EPFL

Advanced Python

Practical Programming in Chemistry

Prof. Philippe Schwaller

Computer setup, all ok?



Programming language to communicate with computer system.
Super useful to know a few commands.

Basic command line usage in bash

Visual Studio Code

A better text editor to handle Python code.

Install VS Code as interactive development environment

General purpose programming language.
Create scripts, programs, webapps, …

ANACONDA®

Install Python with Conda
(main course programming language)

A platform to share, track, and store code.

GitHub

Creating a GitHub account
and getting familiar with basic commands

This lecture

- Writing structured code
  - Function
  - Classes
- Errors
- How to read and write files

Functions in Python

Function example

```
def function_name(parameters):
    """Docstring explaining the function."""
    # Code block
    return result
```

Function name

Parameters

Documentation string. Note the multiline comment with “””

```
def calculate_molarity(moles, volume_liters):
    """Calculate the molarity of a solution.

    Parameters:
    - moles: Number of moles of solute.
    - volume_liters: Volume of solution in liters.

    Returns:
    The molarity of the solution.
    """
    return moles / volume_liters

# Example usage
molarity = calculate_molarity(0.5, 1.0)
print(f"Molarity: {molarity} M")
```

The return statement

Write manageable and efficient code – why functions?

- Without functions, you write the same code over and over.

```python
# Without functions – repetitive and hard to maintain
student1_scores = [85, 92, 78, 90]
student1_total = 0
for score in student1_scores:
    student1_total += score
student1_average = student1_total / len(student1_scores)

student2_scores = [79, 88, 92, 85]
student2_total = 0
for score in student2_scores:
    student2_total += score
student2_average = student2_total / len(student2_scores)
```

Clear naming makes code, easier to understand.

```python
def calculate_average(scores):
    total = 0
    for score in scores:
        total += score
    return total / len(scores)

# Now it's much cleaner and easier to use
student1_average = calculate_average([85, 92, 78, 90])
student2_average = calculate_average([79, 88, 92, 85])
```

- Functions organise the code, and let you breakdown complex problems into smaller, more manageable pieces.

*Think of **functions like recipes in a cookbook**. Instead of writing out all the steps to make bread every time you want to bake it, you can just refer to the bread recipe.*

*Functions work the same way - they're **reusable instructions that you can call whenever you need them**.*

Example: temperature conversion

Triple quotes for multiline documentation string.

default value

```python
def convert_temperature(temp, scale_to='C'):
    """Convert temperature between Celsius and Fahrenheit.

    Parameters:
    - temp: The temperature to convert.
    - scale_to: Target temperature scale ('C' for Celsius, 'F' for Fahrenheit)

    Returns:
    The converted temperature.
    """
    if scale_to.upper() == 'C':
        return (temp - 32) * 5/9
    elif scale_to.upper() == 'F':
        return temp * 9/5 + 32
    else:
        raise ValueError("scale_to must be 'C' or 'F'.")

# Example usage -> positional arguments
print(f"100F in Celsius: {convert_temperature(100, 'C')}°C")

# Example usage -> keyword arguments
print(f"0C in Fahrenheit: {convert_temperature(scale_to='F', temp=0)}°F")
```

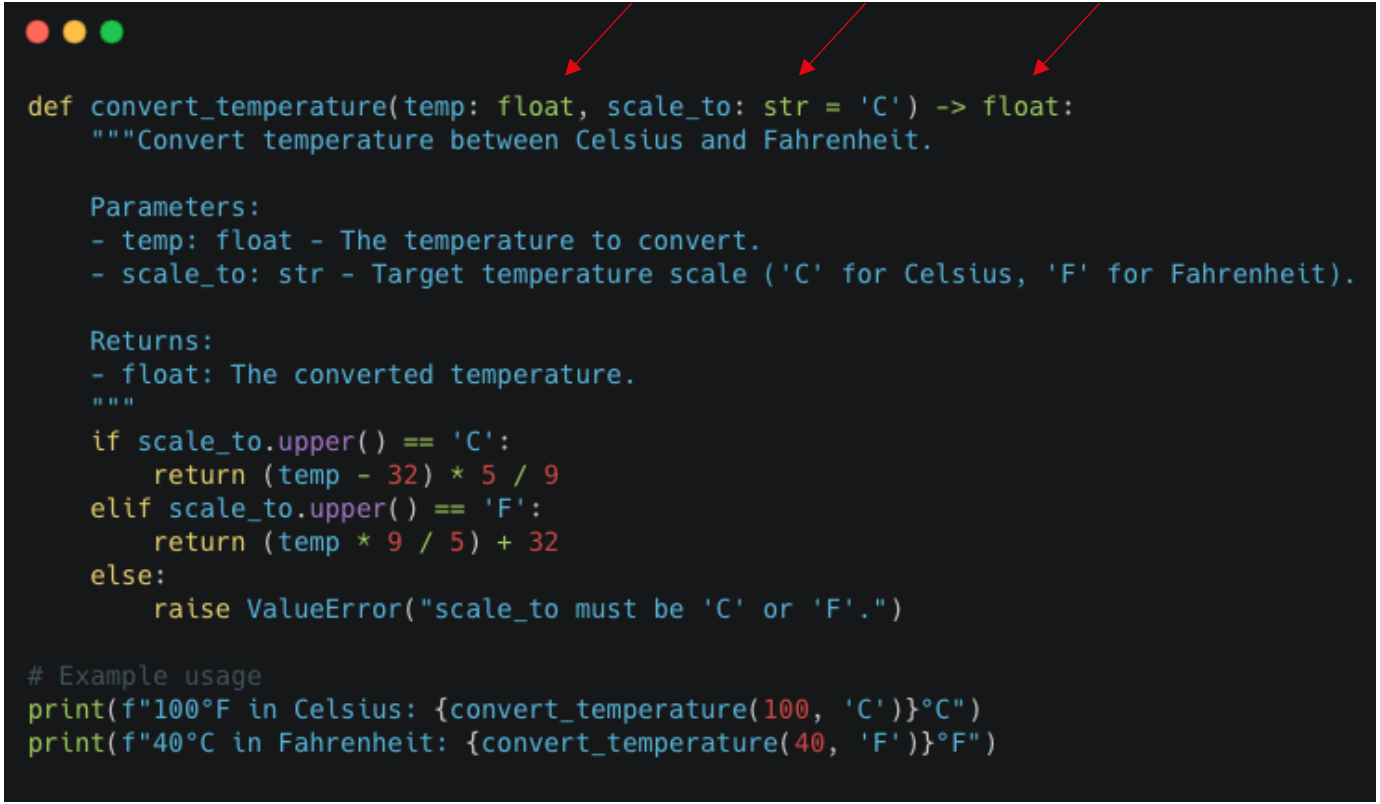
```
In [32]: # Example usage -> positional arguments
    ...: print(f"100F in Celsius: {convert_temperature(100, 'C')}°C")
    ...:
    ...: # Example usage -> keyword arguments
    ...: print(f"0C in Fahrenheit: {convert_temperature(scale_to='F', temp=0)}°F")
100F in Celsius: 37.77777777777778°C
0C in Fahrenheit: 32.0°F
```

```
In [34]: # Example usage -> default value
    ...: print(f"100F in Celsius: {convert_temperature(100)}°C")
100F in Celsius: 37.77777777777778°C
```

different return

Error raised if invalid parameter

Same function with Type hints



```python
def convert_temperature(temp: float, scale_to: str = 'C') -> float:
    """Convert temperature between Celsius and Fahrenheit.

    Parameters:
    - temp: float - The temperature to convert.
    - scale_to: str - Target temperature scale ('C' for Celsius, 'F' for Fahrenheit).

    Returns:
    - float: The converted temperature.
    """
    if scale_to.upper() == 'C':
        return (temp - 32) * 5 / 9
    elif scale_to.upper() == 'F':
        return (temp * 9 / 5) + 32
    else:
        raise ValueError("scale_to must be 'C' or 'F'.")

# Example usage
print(f"100°F in Celsius: {convert_temperature(100, 'C')}°C")
print(f"40°C in Fahrenheit: {convert_temperature(40, 'F')}°F")
```

In Python, types of variables are determined dynamically. But **with type hints** you can let the computer/human know beforehand what **the expected variable type** would be.

Reusing functions

- Imagine you need the same function multiple times
  - In the same notebook → define it at the beginning of the notebook
  - In multiple notebooks → define the function in a ***.py** file

Note: The "*" means any string.

- For the conversion example, you could write it into a "temperature_conversion.py" file in the same folder as your notebook, and then import it

```
In [35]: from temperature_conversion import convert_temperature
```

→ The **advantage** is that if you **modify/improve the function**, you have to **do it once**. Also, it allows you to test a single function.

What functions can I access in my code?

- **Built-in functions** (do not need to be imported), e.g.: *print(), len(), int(), str()*
- **User-defined Functions** (defined using the *def* keyword)

```
def my_function(param1, param2):
    # Function body
    return param1 + param2
```

- **Functions imported from .py Files** (you can import function from .py files in same folder)
  - In “properties.py” you define “def get_molecular_weight(molecule)”
  - In ”analyze_molecules.ipynb”, in the same folder, you import “from properties import get_molecular_weight”, and then you can use the function.
- **Functions imported from installed packages** (e.g. Numpy)

Classes in Python

Other ways to write cleaner code – classes

- Imagine you want to design a digital periodic table
- A **class** is like a template or blueprint of **an element**
- Every element would be an instance (object) of that class
- Each element has a **set of properties**: name, symbol, atomic_number, …, in programming, we call those **attributes**.
- And you could have **methods** (functions) that an element can perform.

```
class Element:
    def __init__(self, name, symbol, atomic_number):
        self.name = name
        self.symbol = symbol
        self.atomic_number = atomic_number

    def describe(self):
        print(f"{self.name} ({self.symbol}), Atomic Number: {self.atomic_number}")
```

Initialisation - __init__ function

__init__ stands for **initialisation**

- The __init__ function only runs once when the object is created.

```
class Element:
    def __init__(self, name, symbol, atomic_number):
        self.name = name
        self.symbol = symbol
        self.atomic_number = atomic_number

    def describe(self):
        print(f"{self.name} ({self.symbol}), Atomic Number: {self.atomic_number}")
```

```
hydrogen = Element("Hydrogen", "H", 1)
```

Inheritance

```python
class Element:
    def __init__(self, name, symbol, atomic_number):
        self.name = name
        self.symbol = symbol
        self.atomic_number = atomic_number

    def describe(self):
        print(f"{self.name} ({self.symbol}), Atomic Number: {self.atomic_number}")

class NobleGas(Element):
    def __init__(self, name, symbol, atomic_number, standard_state="gaseous"):
        super().__init__(name, symbol, atomic_number)
        self.standard_state = standard_state

    def describe_state(self):
        print(f"{self.name} is in its {self.standard_state} state at room temperature.")
```

This will run the __init__ of the parent class.

```python
# Example usage
helium = NobleGas("Helium", "He", 2)
helium.describe()
helium.describe_state()
```

Why We Use Classes?

- **Encapsulation:** Classes bundle **data (attributes)** and **methods/functions** that operate on the data into a **single entity (object). --> better structure**
- **Inheritance:** Classes allow for the creation of a **new class that inherits attributes and methods from an existing class**. --> less rewriting
- **Data Abstraction:** Classes can provide a **simple interface to complex systems.**.
- **Modeling Real-world Problems:** Object-oriented programming allows you to model real-world entities and relationships in a more natural and intuitive way.
  In chemistry, for example, you can model molecules, atoms, reactions, and laboratory equipment as objects with specific attributes and behaviors.

Because it makes writing complex code bases easier.

- This is just for you to know what it is, when you come across a class in Python code you look at.

In short: Functions and classes

- Both aim to **build complexity from simpler structures**
- **Functions:** One place, to define **clear inputs and outputs** that can be reused in multiple parts of your code
- **Classes:** Instead of rewriting every chemical compound/element object from scratch, you can **define common functionality shared by all** members of the class.

Errors and exceptions

Errors in Python

- Python has several **built-in errors** that can occur during the execution of a program. Understanding these common errors and exceptions is crucial for debugging and writing robust Python code.

```
if x == 1
    print("x is 1")
```

```
In [36]: if x == 1
    ...:     print("x is 1")
  Cell In[36], line 1
    if x == 1
             ^
SyntaxError: invalid syntax
```

```
In [37]: if x == 1:
    ...:     print("x is 1")
    ...:
---------------------------------------------------------------------------
NameError
Cell In[37], line 1
----> 1 if x == 1:
      2     print("x is 1")

NameError: name 'x' is not defined
```

```
In [38]: x =1
    ...: if x == 1:
    ...:     print("x is 1")
    ...:
x is 1
```

Other common errors

```
# Example of TypeError
"2" + 2  # Attempting to add a string and an integer
```

```
Cell In[39], line 2
      1 # Example of TypeError
----> 2 "2" + 2  # Attempting to add a string and an integer

TypeError: can only concatenate str (not "int") to str
```

```
In [40]: 2 + "2"
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[40], line 1
----> 1 2 + "2"

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

```
my_list = [1, 2, 3]
print(my_list[3])
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[41], line 3
      1 # Example of IndexError
      2 my_list = [1, 2, 3]
----> 3 print(my_list[3])  # There is no index 3 in this list

IndexError: list index out of range
```

```
my_dict = {"name": "Alice"}
print(my_dict["age"])
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Cell In[43], line 2
      1 my_dict = {"name": "Alice"}
----> 2 print(my_dict["age"])

KeyError: 'age'
```

Other common errors

```
# Example of AttributeError
"hello".append('world')  # String object has no attribute 'append'
```

```
# Example of ValueError
int("xyz")  # Trying to convert a string that does not contain numbers to an integer
```

```
# Example of ZeroDivisionError
1 / 0
```

```
# Example of FileNotFoundError
with open('nonexistent_file.txt') as f:
    read_data = f.read()
```

```
# Example of ImportError
from math import non_existent_function
```

How to handle errors/exceptions – try/except

- **Exceptions in Python** are errors detected during execution that **interrupt** the normal **flow of a program**.
  They **can be caught and handled** to prevent the program from crashing and to provide more **informative error messages** or alternative solutions.

- **Exception Handling:** Python uses **try, except, else,** and **finally** blocks to handle exceptions. This allows programmers to write code that can gracefully deal with unexpected errors.
- **Raising Exceptions:** Beyond the built-in exceptions, Python allows raising custom exceptions using the **raise statement**. This is useful for signaling error conditions in your code in a way that's consistent with Python's error handling model.

Try/except synthax

```
try:
    # Code block where you suspect an exception might occur
    pass
except SomeException:
    # Code that runs if SomeException occurs
    pass
else:
    # Code that runs if no exceptions occur within the try block
    pass
finally:
    # Code that runs no matter what, used for cleanup
    pass
```

```
try:
    result = 10 / 0
except ZeroDivisionError:
    print("You can't divide by zero!")
else:
    print("Division successful!")
finally:
    print("This block executes no matter what.")
```

```
x = some_function()

try:
    result = 10 / x
except ZeroDivisionError:
    print("You can't divide by zero!")
else:
    print("Division successful!")
finally:
    print("This block executes no matter what.")
```

raise statement

```python
def convert_temperature(temp, scale_to='C'):
    """Convert temperature between Celsius and Fahrenheit.

    Parameters:
    - temp: The temperature to convert.
    - scale_to: Target temperature scale ('C' for Celsius, 'F' for Fahrenheit)

    Returns:
    The converted temperature.
    """
    if scale_to.upper() == 'C':
        return (temp - 32) * 5/9
    elif scale_to.upper() == 'F':
        return temp * 9/5 + 32
    else:
        raise ValueError("scale_to must be 'C' or 'F'.")

# Example usage -> positional arguments
print(f"100F in Celsius: {convert_temperature(100, 'C')}°C")

# Example usage -> keyword arguments
print(f"0C in Fahrenheit: {convert_temperature(scale_to='F', temp=0)}°F")
```

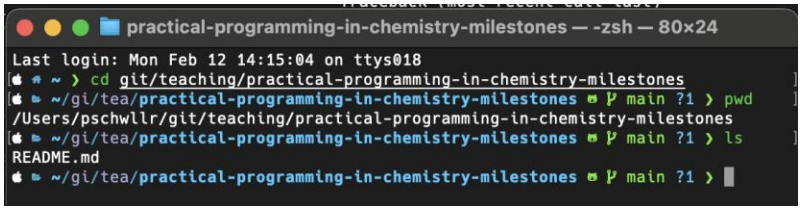
```
In [44]: convert_temperature(0, "A")
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[44], line 1
----> 1 convert_temperature(0, "A")

Cell In[29], line 16, in convert_temperature(temp, scale_to)
     14     return temp * 9/5 + 32
     15 else:
---> 16     raise ValueError("scale_to must be 'C' or 'F'.")

ValueError: scale_to must be 'C' or 'F'.
```

How to access files on your computer?

- On key challenge is that paths on Windows and MacOS/Linux are different.
- There are backslashes (\) on Windows and forward slashes (/) on Unix-like systems (macOS, Linux).
  - Windows: C:\Users\Username\Documents\project\myfile.txt
  - Mac/Linux: /home/username/Documents/project/myfile.txt

- `C:` is the drive letter.
- `\Users\Username\Documents\project\` is the path to the directory containing the file.
- `myfile.txt` is the name of the file.

- `/` is the root directory.
- `home/username/Documents/project/` is the path to the directory containing the file.
- `myfile.txt` is the name of the file.



Pathlib or os for compatibility

```python
from pathlib import Path

# For a relative path
p = Path('some_directory/myfile.txt')

# For an absolute path
home = Path.home()
config_path = home / '.config' / 'app'

# Opening a file
with open(p, 'r') as f:
    content = f.read()
```

```python
import os

path = os.path.join('some_directory', 'myfile.txt')
```

Relative paths are preferred!

Note: Use forward slashes (/). Python will automatically convert these to the appropriate separator for the operating system on which it's running.

Avoid: Hardcoded absolute paths like “Users/username/some_directory”

Reading files in Python

- The key function for working with files in Python is **open()**, which is used to open a file and **returns a file object**. This file object can then be used to read from or write to the file.

Issue: if error is raised, file is never closed.

```
with open('example.txt', 'r') as file:
    content = file.read()
    print(content)
```

Handles the file-closing, when something goes wrong. Always use **with.**

Different modes for open()

Opening a File

To open a file in Python, we use the open function. The `open` function returns a file object and is most commonly used with two arguments: `open(filename, mode)`.

- The `filename` is the name (and the path if the file is not located in the same directory as the Python script) of the file you want to open.
- The `mode` argument is a string that defines which mode you want to open the file in:
  - `"r"`: read mode (default)
  - `"w"`: write mode, for overwriting the contents of a file
  - `"x"`: exclusive creation mode, for creating a new file and failing if it already exists
  - `"a"`: append mode, for appending data to an existing file
  - `"b"`: binary mode
  - `"t"`: text mode (default)

Reading from a file

- `read()` : This reads the entire file.
- `readline()` : This reads a file line by line.
- `readlines()` : This reads all the lines and returns them as a list of strings.

```
In [46]: less compounds.txt
Water
Methane
Ammonia
Hydrogen Peroxide
Acetic Acid
compounds.txt lines 1-5/5 (END)
```

```
In [47]: with open('compounds.txt', 'r') as file:
    ...:     content = file.read()
    ...:     print(content)
    ...:
Water
Methane
Ammonia
Hydrogen Peroxide
Acetic Acid
```

```
In [49]: with open('compounds.txt', 'r') as file:
    ...:     line = file.readline()
    ...:     print(line)
    ...:
Water
```

\n → end of line character (or \r\n)

```
In [52]: with open('compounds.txt', 'r') as file:
    ...:     lines = file.readlines()
    ...:     print('Water' in lines)
    ...:
False
```

Why is this the case?

```
In [53]: with open('compounds.txt', 'r') as file:
    ...:     lines = file.readlines()
    ...:     print(lines)
    ...:
['Water\n', 'Methane\n', 'Ammonia\n', 'Hydrogen Peroxide\n', 'Acetic Acid\n']
```

End of line character

```
In [54]: with open('compounds.txt', 'r') as file:
    ...:     lines = [line.strip() for line in file.readlines()]
    ...:     print(lines)
    ...:
['Water', 'Methane', 'Ammonia', 'Hydrogen Peroxide', 'Acetic Acid']
```

Common trick!

- Uses a list comprehension (last week) and the **strip()**, which removes whitespace around a string.

Write files

```
with open('chemicals.txt', 'w') as file:
    file.write('New line in file.')
```

```
In [56]: compounds
Out[56]: ['Water', 'Methane', 'Ammonia', 'Hydrogen Peroxide', 'Acetic Acid']

In [57]: with open('compounds.txt', 'w') as file:
    ...:     for compound in compounds:
    ...:         file.write(compound)
    ...:
```

```
In [58]: less compounds.txt
WaterMethaneAmmoniaHydrogen PeroxideAcetic Acid
```

```
In [61]: with open('compounds.txt', 'w') as file:
    ...:     for compound in compounds:
    ...:         file.write(compound + '\n')
    ...:

In [62]: less compounds.txt
Water
Methane
Ammonia
Hydrogen Peroxide
Acetic Acid
```

```
In [64]: with open('compounds.txt', 'w') as file:
    ...:     file.write('\n'.join(compounds))
    ...:

In [65]: less compounds.txt
Water
Methane
Ammonia
Hydrogen Peroxide
Acetic Acid
```

1 liner, using **join()**

```
In [66]: '\n'.join(compounds)
Out[66]: 'Water\nMethane\nAmmonia\nHydrogen Peroxide\nAcetic Acid'
```

Today's exercises

- Writing and reading files
- Functions
- Error handling
- Classes