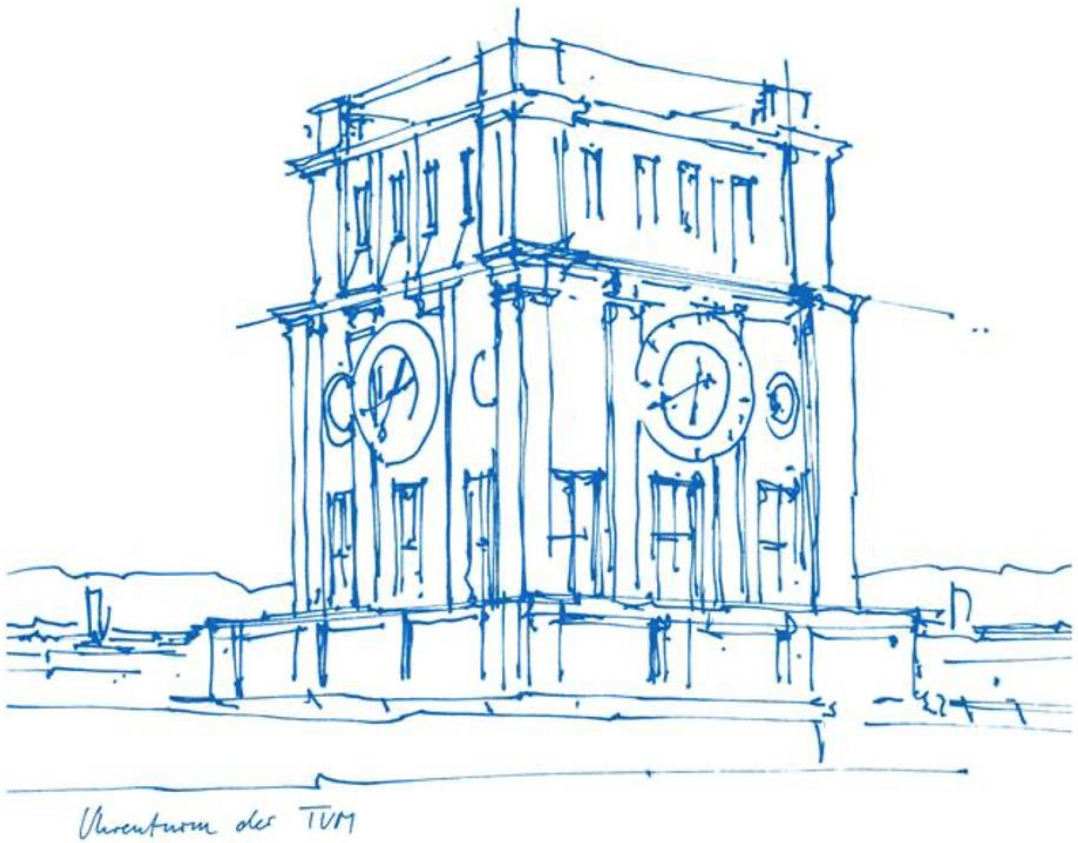
School of Life Sciences & School of Management
Technical University of Munich

Programming in Python for Business and Life Science Analytics (MGT001437)

Day 1 – 19th Nov 2025

Winter Term 2025/2026

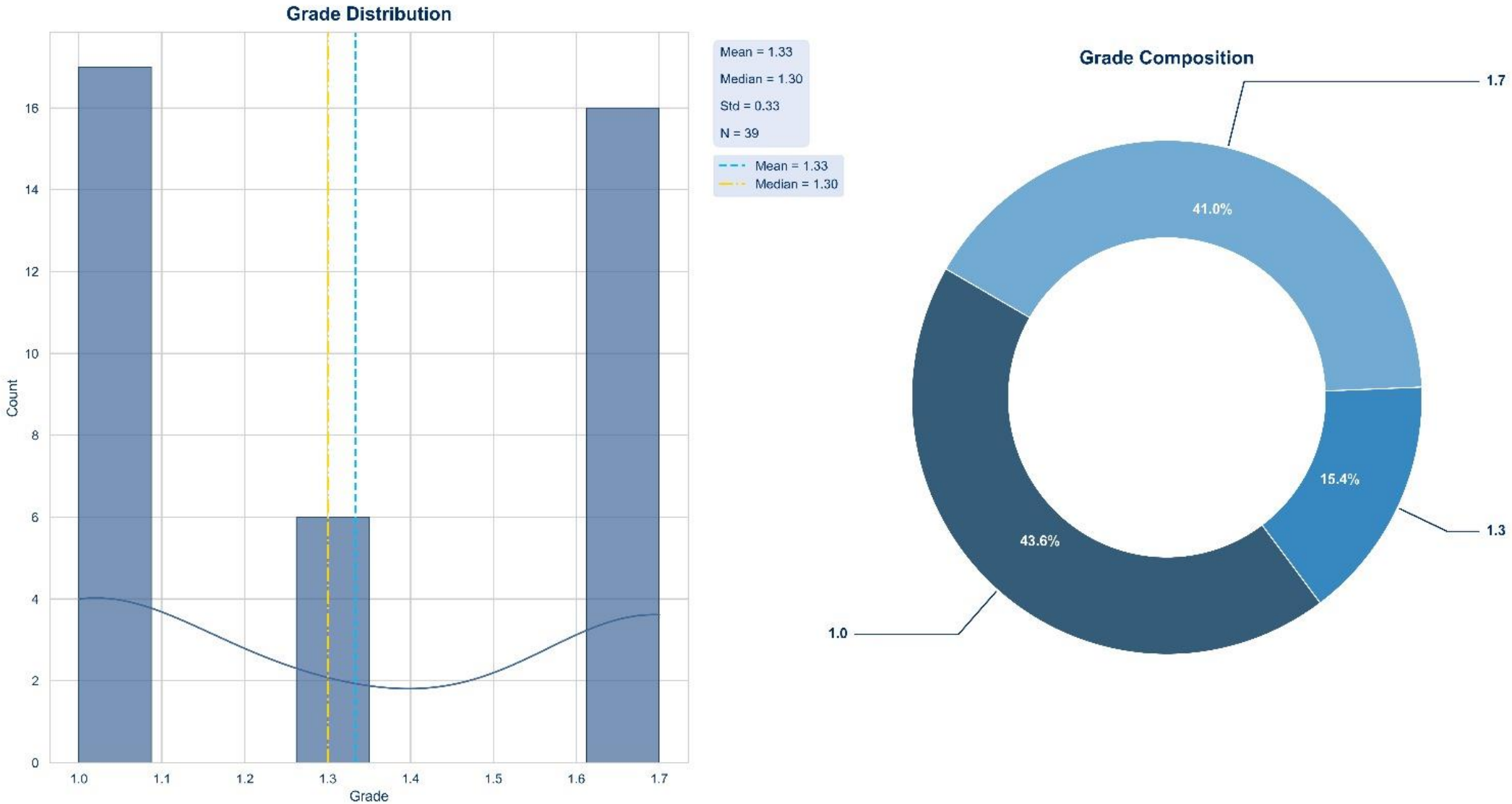


Yanfei Shan | Munich Data Science Institute |
Sustainability Assessment of Food and Agricultural Systems

9:00 – 10:00

Introduction, Installation and Setup

Grade Analysis Report (24SS and 24/25WS)



Why Python?

The Nobel Prize in Physics 2024

John Hopfield

"for foundational discoveries and inventions that enable machine learning with artificial neural networks"

John Hopfield. Ill. Niklas Elmehed © Nobel Prize Outreach

Geoffrey Hinton

"for foundational discoveries and inventions that enable machine learning with artificial neural networks"

Geoffrey Hinton. Ill. Niklas Elmehed © Nobel Prize Outreach

- **Leading language for building neural networks (AI)**
- High-Level Language with Simple Syntax
- **Exceptional for Data Processing and Analysis**
- Great Support for Web Development and Automation (Django and Flask)
- Highly Versatile with Strong Community Support
- Strong Integration Capabilities

Before we start, we need a **code editor** – VS Code

We use Python to *write programs*, and we use VS Code to *write Python code more efficiently.*

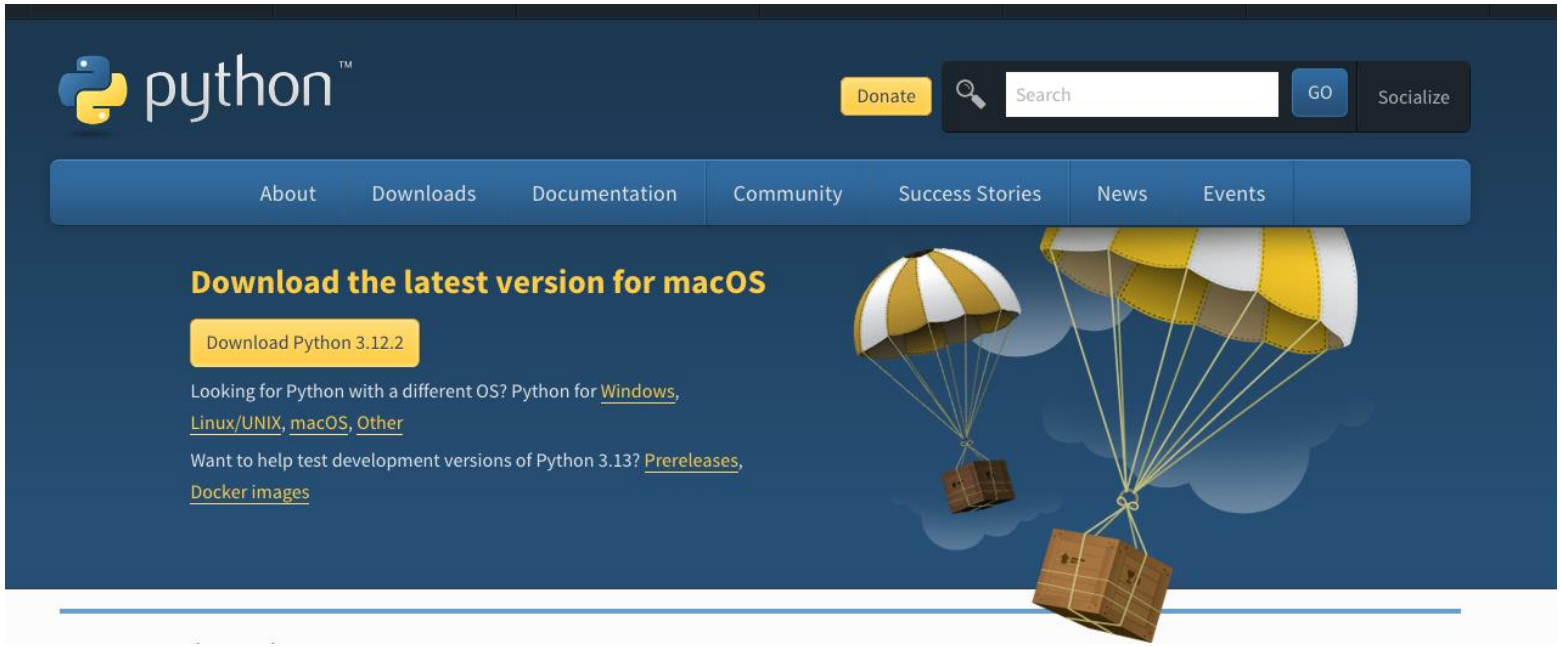
- **Visual Studio Code** – Free integrated development environment(IDE) made by Microsoft for Windows, macOS and Linus.

- Features – Supports many languages(eg. Javascript, html, css..) and extensions, debugging, syntax highlighting, intelligent code completion, snippets, code refactoring, and embedded Git.

- Productivity – Lightweight and fast, easy to set up and use.

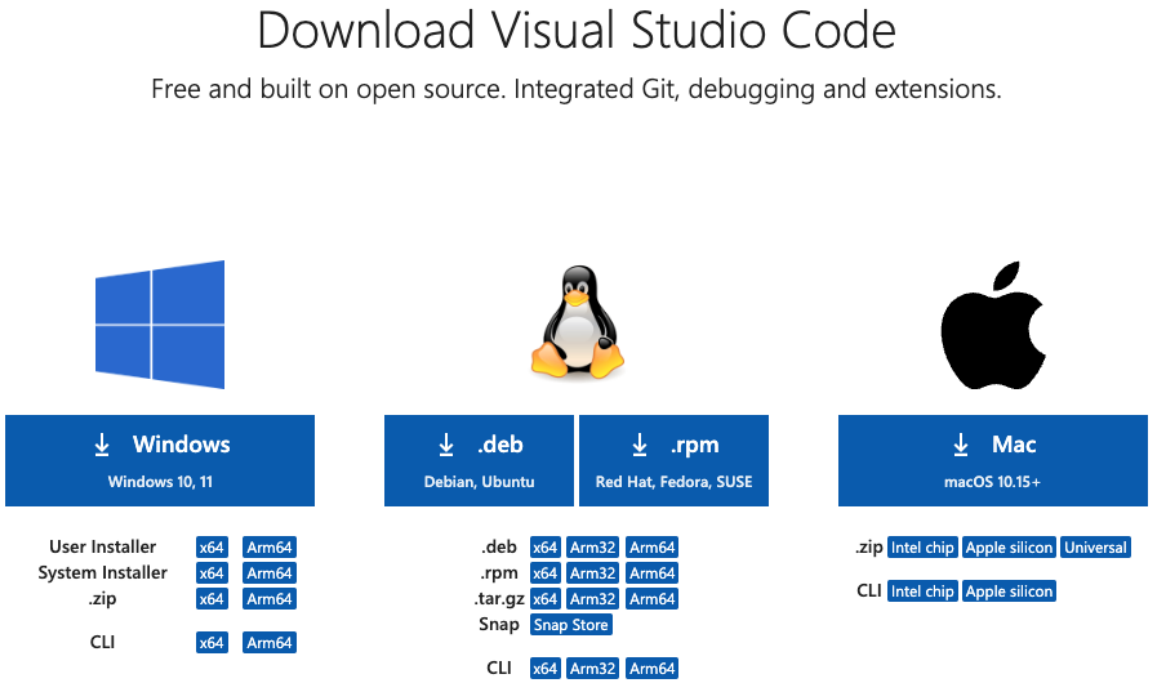
Installing Python and VS Code

- Download Python from https://www.python.org/downloads/
- Download VS Code from https://code.visualstudio.com/download
- Instruction video for **Windows:** https://www.youtube.com/watch?v=cUAK4x_7thA&ab_channel=HarsivoEdu
- Instruction video for **Mac:** https://www.youtube.com/watch?v=NirAuEAblvo&ab_channel=SonarSystems



Installing Python and VS Code

- Download Python from https://www.python.org/downloads/
- Download VS Code from https://code.visualstudio.com/download
- Instruction video for **Windows:** https://www.youtube.com/watch?v=cUAK4x_7thA&ab_channel=HarsivoEdu
- Instruction video for **Mac:** https://www.youtube.com/watch?v=NirAuEAblvo&ab_channel=SonarSystems



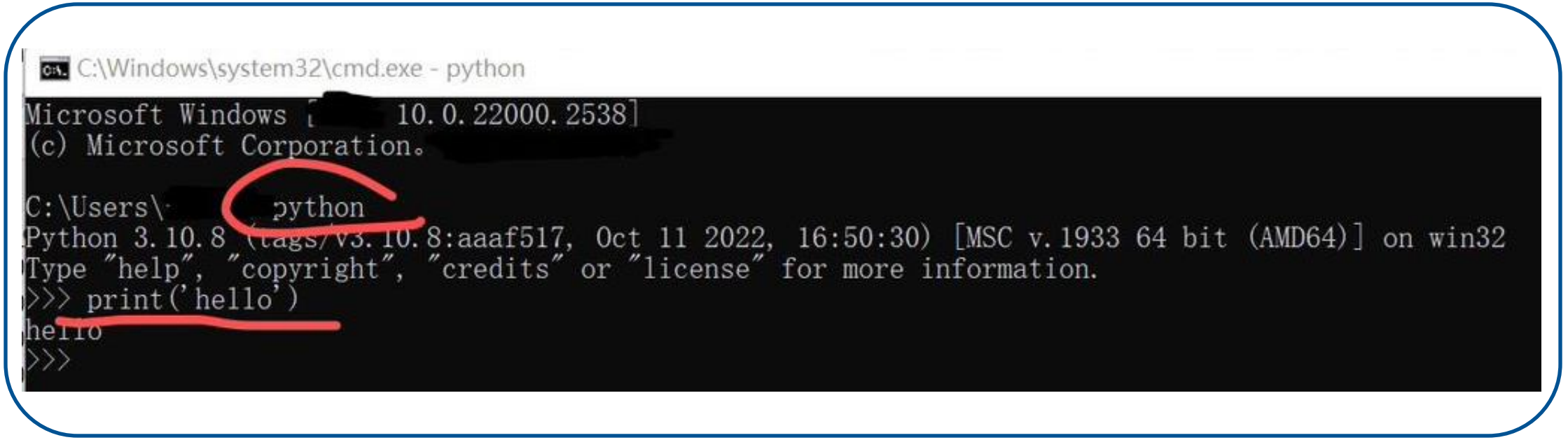
How to run Python in Terminal? - MacOS

1. Use Command + Space shortcut to bring up Spotlight search window.
2. Type Terminal.
3. Double-click on Terminal in search results to open it.

```
faye@tangxiaoyedeMacBook-Pro ~ % python3 --version
Python 3.9.6
faye@tangxiaoyedeMacBook-Pro ~ % python3
Python 3.9.6 (default, Aug  5 2022, 15:21:02)
[Clang 14.0.0 (clang-1400.0.29.102)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print('hello')
hello
>>>
```

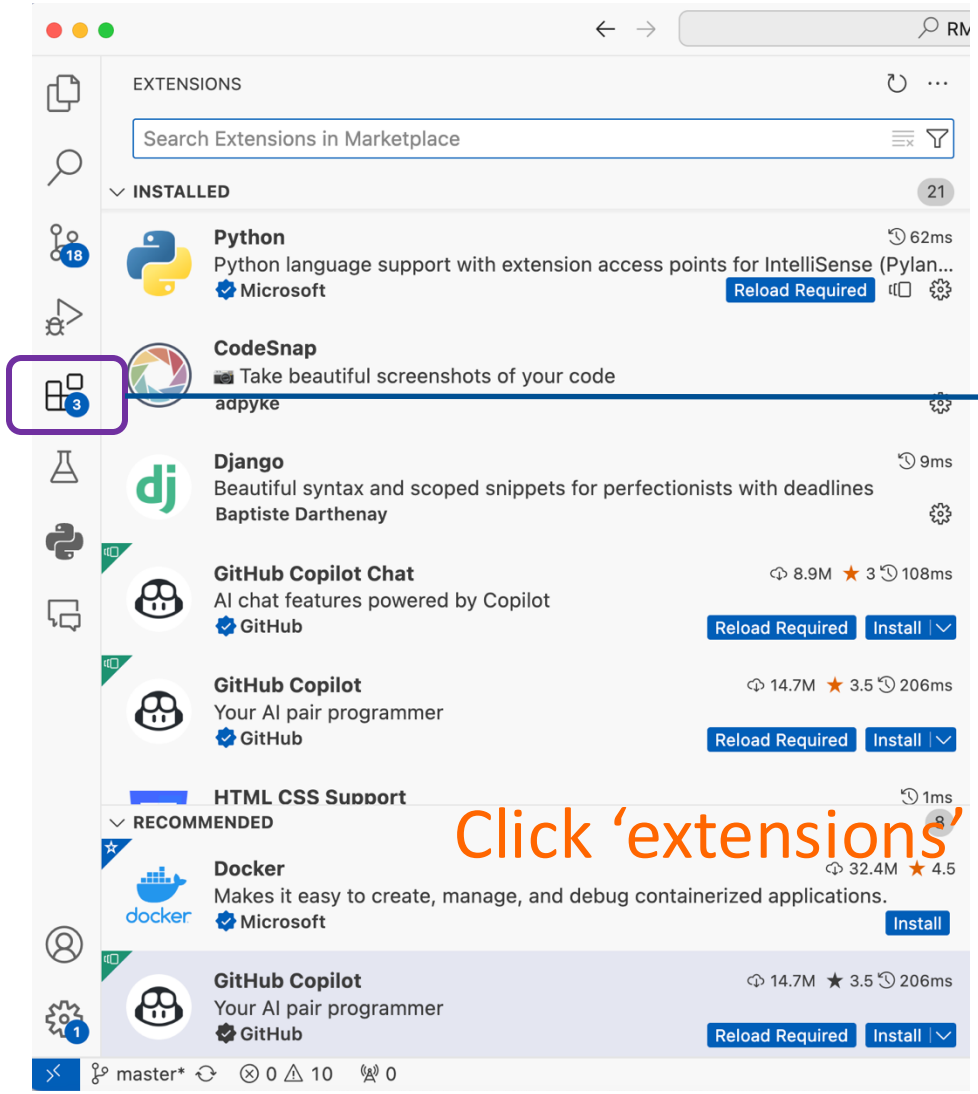
How to run Python in Terminal? - Windows

Shortcut to open the terminal: Command + ‘R’

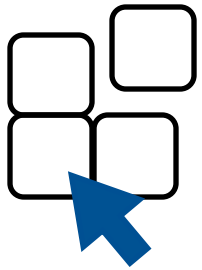


The shell isn’t a great way to do anything
more than type in one line of code line by line

VS Code Setup - Extensions



Click ‘extensions’



Click on “Extensions”

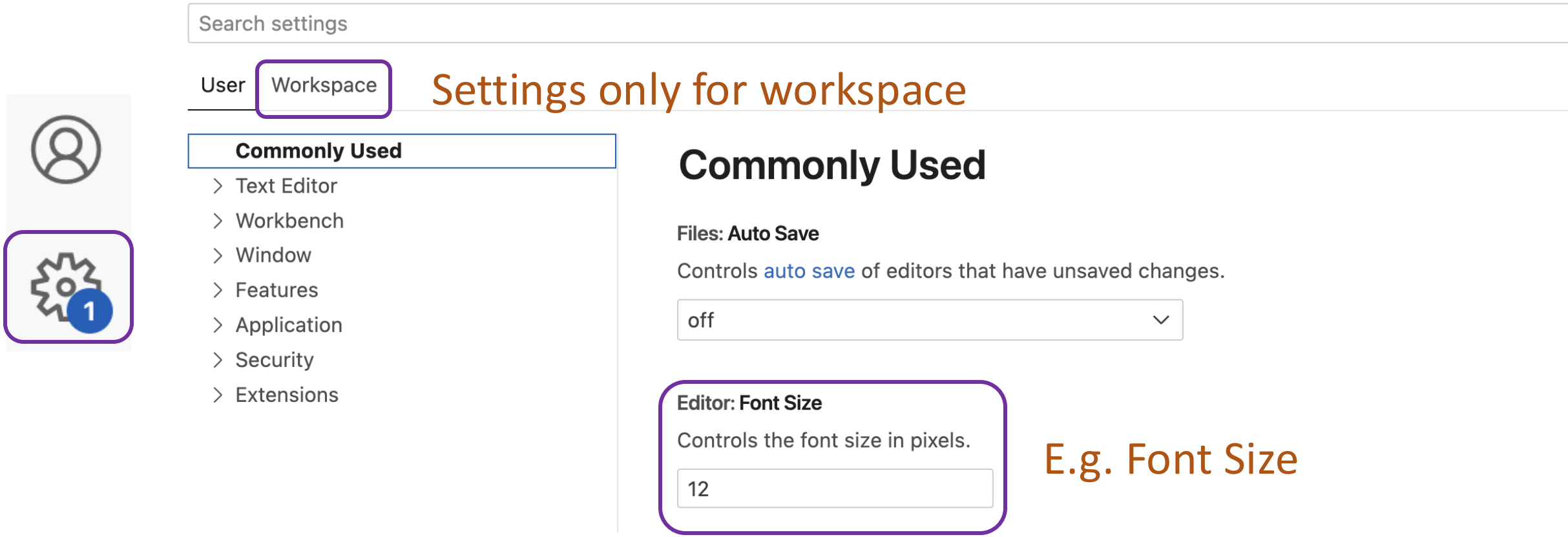
Useful extensions: (better to install)

- Python
- Python Environment Manager
- Python Debugger
- Pylance
- Jupyter
- Jupyter Cell Tags
- Jupyter Keymap
- Todo Tree
- Office Viewer
- Matlab / R
- Remote - SSH

VS Code Setup - Settings

Settings,

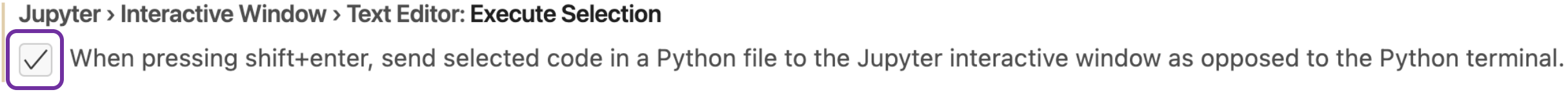
Themes (change the colour of the background) …



VS Code Setup - jupyter se (Must Do)

→ **Search “jupyter se” here**

→ **Check the box**



I’ll explain you why

Github

GitHub is an online platform for version control and collaborative software development, allowing developers to host, review, and manage code projects. It's built on Git, a version control system that tracks changes in code, making it easier to collaborate and maintain code integrity across teams. With GitHub, users can create repositories to store code, document projects, and track issues. Developers can work together by branching, committing, and merging code changes, ensuring that multiple contributors can efficiently contribute to a project. GitHub also supports open-source collaboration, making it a central hub for sharing and improving code across the global developer community.

Github desktop: https://desktop.github.com/download/

Github readme example: https://bulldogjob.com/readme/how-to-write-a-good-readme-for-your-github-project

Google Colab

- Introduction:

https://aguaclara.github.io/aguaclara_tutorial/colab/colab-introduction.html

- Markdown:

https://colab.research.google.com/drive/15yYapOXlOLOEYMttz8v1Vb5odRkkJdhO

- Python:

https://colab.research.google.com/drive/15yYapOXlOLOEYMttz8v1Vb5odRkkJdhO

Anaconda

VS Code is only a **code editor** — a place to *write* code, not to *manage or run* Python environments.

So even if you install the Python extension in VS Code, you still need a **real Python environment** underneath.

This is where **Anaconda** becomes extremely useful.

- Anaconda is a popular distribution of Python and R programming languages for scientific computing, data science, machine learning applications, and more.
- Install: https://docs.anaconda.com/free/anaconda/install/index.html

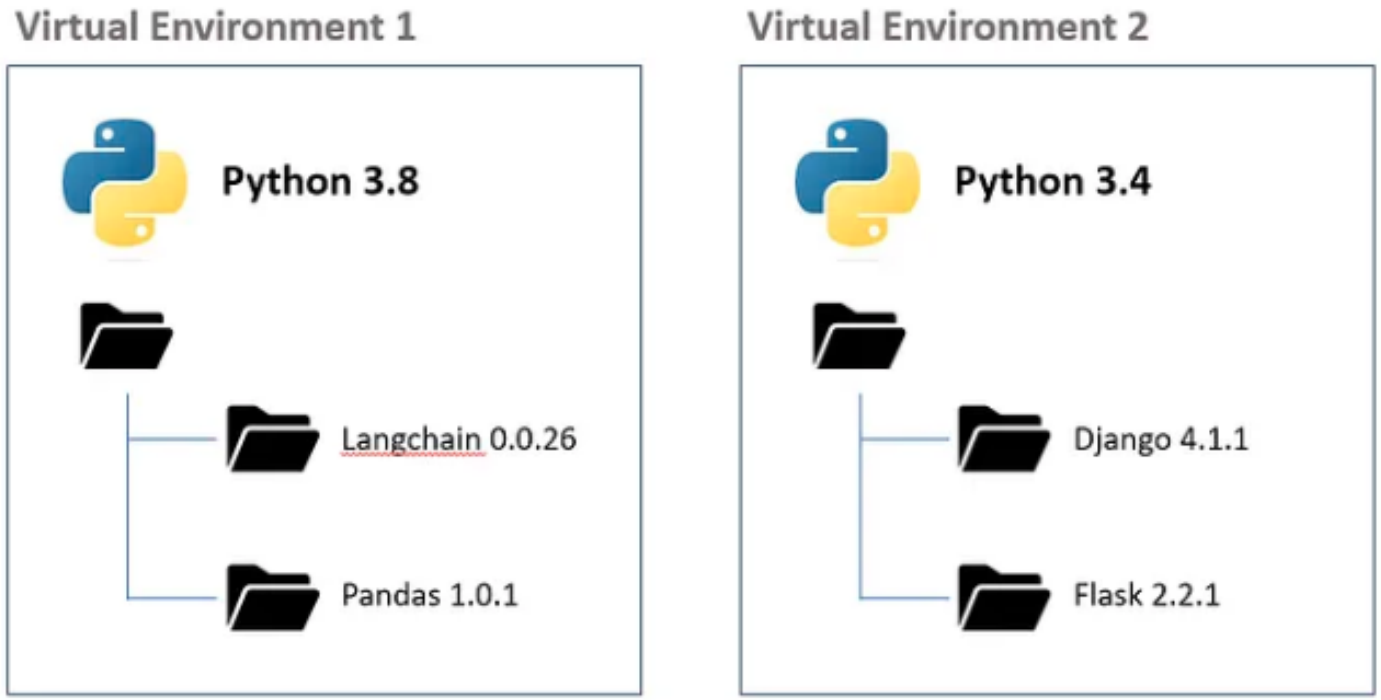
What is an Environment?

A **Python environment** is a **separate workspace** that contains:

- a specific **Python version**
- its own installed **libraries**
- its own settings

A **library** (also called “package” or “module”) is a **collection of pre-written code** that helps you do things faster.

Why libraries matter:
- You don’t need to reinvent the wheel
- You can do data analysis, machine learning, plotting, etc. in 1–2 lines of code
- The entire Python ecosystem is built on libraries



Material: https://www.youtube.com/watch?v=KxvKCSwlUv8

We will learn numpy, pandas, scikit-learn, torch....

10:00 – 11:00

Project pipeline and project management (VScode, github)

Create virtual environment (1)

1. Dependency Management

•**Isolated Environments**: A virtual environment allows you to create an isolated environment for each Python project, ensuring that dependencies and packages installed for one project do not interfere with others.
•**Version Conflicts**: Projects often require different versions of the same library. For example, one project might need Django 2.2, while another might require Django 4.0. Virtual environments let you manage these versions independently without conflicts.

2. Reproducibility

•**Consistent Development and Production Environments**: By using virtual environments, you can replicate the exact environment where your code runs. This is crucial for ensuring that code developed on your local machine works the same way when deployed to a server or shared with others.
•**Requirements File**: You can create a requirements.txt file that lists all the dependencies of your project, allowing anyone who clones your project to create an identical environment with pip install -r requirements.txt.

3. Simplified Package Management

•**Avoiding Global Installations**: Installing packages globally (i.e., system-wide) can lead to clutter and potential conflicts with other software. Virtual environments provide a clean slate for installing only the packages needed for a specific project.
•**Easy Cleanup**: If you need to uninstall or remove a project, you only need to delete the virtual environment, keeping your system clean and organized.

Create virtual environment (1)

- **conda create -n (custom env name) (python-version) (packages)**
- **conda activate (custom env name)**
- conda deactivate
- conda install (package name)
- conda env list
- conda env remove --name (custom env name) –all
- conda list
- **conda update (package) Update a package**
- **conda remove (package)**

Create virtual environment (2)

- check which packages have not updated:

conda list --outdated

- conda update –all

- pip list --outdated
- pip install --upgrade (packagename)

Pip vs Conda

- **pip** is the package installer for Python. It installs packages from the Python Package Index (PyPI), which contains a vast array of Python packages from the community. It's a more general-purpose tool. pip installs dependencies as specified by package maintainers; however, it does not always ensure that these dependencies work harmoniously (i.e., it does not prevent dependency conflicts).

- **conda** is a package and environment management system that comes with Anaconda. It can install Python packages as well as packages from other languages like R, Scala, etc. Conda packages are typically from the Anaconda repository, which hosts packages specifically built and maintained by the Anaconda team. conda attempts to analyze all of the dependencies of the packages you want to install to ensure that they work together well, which minimizes the risk of conflicts.

If ‘pip’ is not installed

- curl https://bootstrap.pypa.io/get-pip.py -o get-pip.py
- python get-pip.py

Import

import means:

Use code that is written somewhere else.

It loads functions, classes, and variables from another file or package into your current script.

Python encourages code reuse — you don't write everything from scratch.

Scope 1: External libraries or **built-in** libraries.

Python looks for them in:

1. **Your environment's site-packages** (installed via pip/conda)

2. **Python's built-in libraries**

If it can't find them →

ModuleNotFoundError: No module named 'xxx'

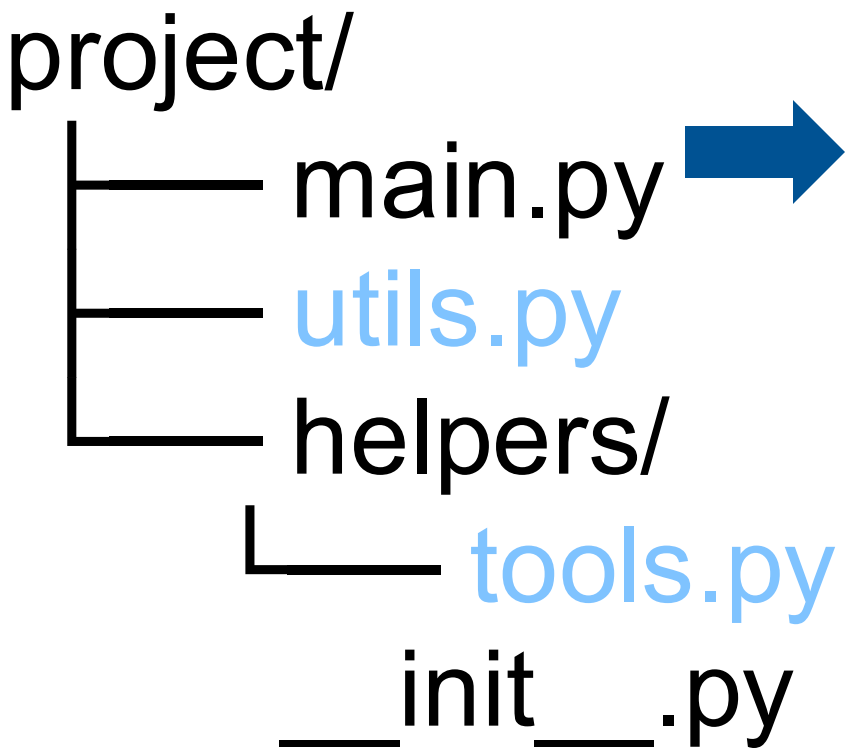
```
from math import sqrt, pi
print(sqrt(16)) # Directly using sqrt without the module prefix
```

```
import pandas as pd
df = pd.DataFrame({'col1': [1, 2, 3]})
```

```
from math import *
print(sqrt(16))
```

```
import math
print(math.sqrt(16))
```

Importing local files




```
import utils
from utils import my_function

from helpers import tools
from helpers.tools import
some_function
```

Python Packages

A package in Python is a collection of related modules organized in a directory hierarchy. It is essentially a way to organize Python code into multiple directories and files to make it easier to manage, understand, and reuse.

Module: A **single Python file** containing code, functions, classes, or variables (e.g., math.py).

Package: A directory that contains a special **__init__.py** file (which can be empty) and one or more module files. The __init__.py file indicates that the directory should be treated as a package.

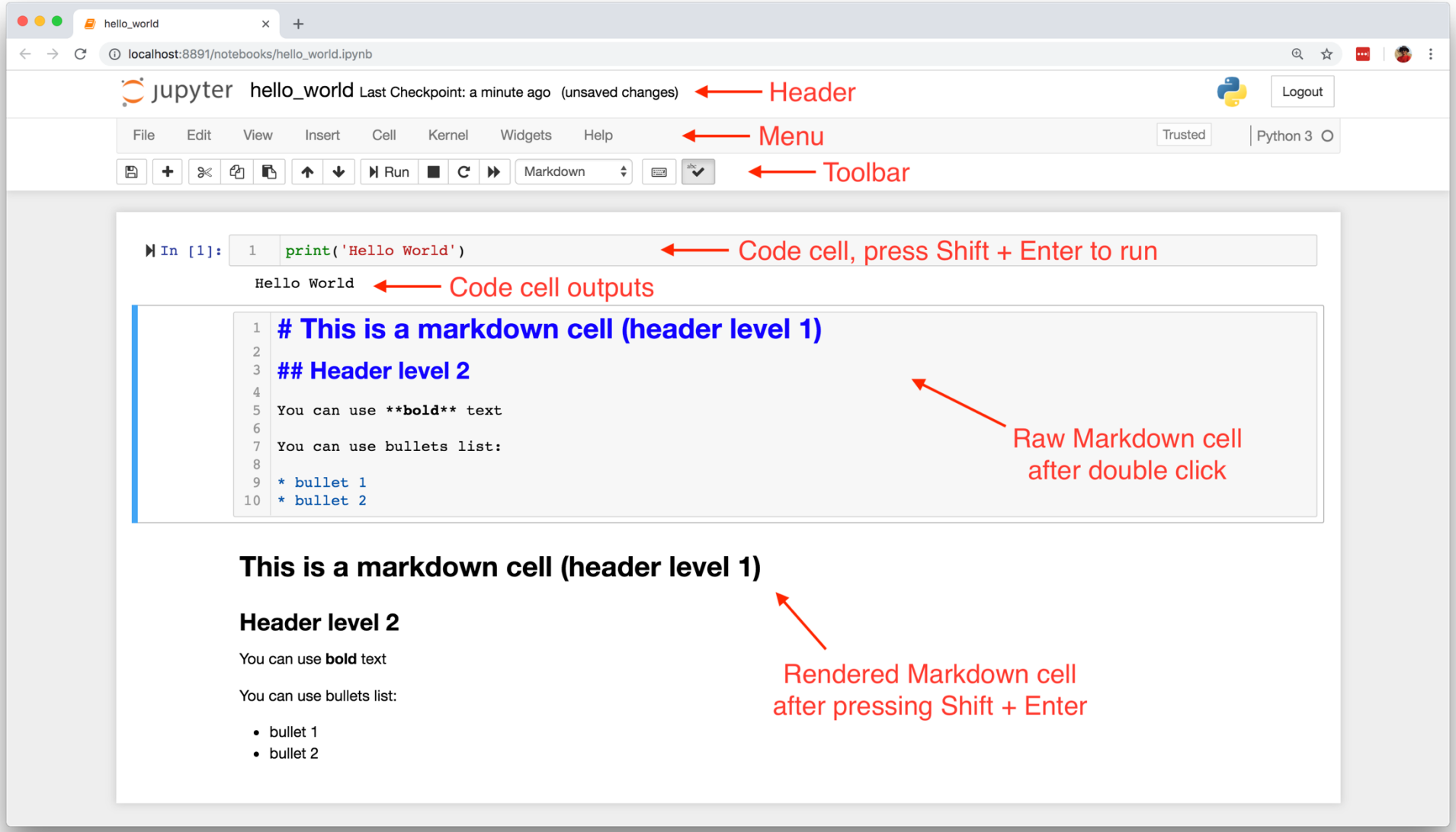
Why Use Packages

- **Organization**: Packages help keep code organized and modular, which is beneficial for large projects.

- **Reusability: Code organized in packages can be reused across multiple projects.**

- **Namespace Management**: Packages create separate namespaces, which help prevent name conflicts between different modules or libraries.

Jupyter Notebook Introduction



Jupyter Notebooks are an essential tool for any data scientist coding in Python.

A more detailed introduction can be accessed via:
https://jupyter.org/try-jupyter/notebooks/?path=notebooks/Intro.ipynb

Jupyter Notebook Introduction

Even though Python scripts (`.py` files) are powerful and essential, **Jupyter Notebook is still one of the most important tools** for data science, machine learning, and GPU-based experiments.

1. Jupyter Notebook is Designed for *Interactive Experimentation*
2. Jupyter Notebook Works Extremely Well in **GPU / HPC / Supercomputer Environments**
3. It Is Safer for GPU Experiments

When to use Notebook:
- exploring data
- trying ideas
- debugging
- training small models
- visualizations
- prototyping
- tuning Hyperparameters

When to use `.py`:
- final model training
- production code
- large-scale pipelines
- modular code structure

In practice, we do this:
Notebook for experiments → convert to `.py` for production.

Jupyter Notebook **Keyboard Shortcuts**

Enter + Shift: Run and move to the next cell

Control + Enter: Run and stay in that cell (If you want to only run that cell/run the cell as many times as you want)

Escape: Escape out of the cell

Key "J": Move to the next cell

Key "K": Move to the upper cell

Key "a": Add a cell above the current cell

Key "b": Add a cell below the current cell

Speed up your day to day workflow!!

Jupyter Notebook **Keyboard Shortcuts**

Key “d” “d”: Delete the cell

Key “M”: Change the cell to “Markdown” mode

Key “Y”: Change the cell to code mode

Command + [: Cancel indentation

Command + shift + “L”: Select all matches of the selected words in the cell

Command + “D”: Select the next match of the selected words in the cell

Command + “S”: Save

11:00 – 12:00

Variables, release memory, sequence types

What Are Variables in Python?

A **variable** is a **name** that stores a **value** in memory.

variables allow us to:
store data
reuse values
make code readable
perform calculations
control program flow
build functions and models

```
>>> width = 5
>>> height = 3
>>> width * height
15
>>> width + _
20
>>> depth
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'depth' is not defined
```

- Variable names can **consist only of letters**, **numbers**, and **underscores (_)**, **and they can't start with numbers**
- Python is **case sensitive**

Good examples: **name**, **age**, **student_id**, **total_sum**, **is_valid**

Wrong examples: 2value, hello world, price$

Good examples:

total_price = 200

student_count = 35

Variables

```
>>> x = 34.5
>>> print(x)
34.5
>>> x = "help"
>>> print(x)
help
>>> print(x * 2)
helphelp
>>> print(x + "!")
help!
```

```
>>> hello_variable = 89
>>> 123_hello_variable = 89
  File "<stdin>", line 1
    123_hello_variable = 89
                       ^
SyntaxError: invalid syntax
>>> hello variable = 89
  File "<stdin>", line 1
    hello variable = 89
                   ^
SyntaxError: invalid syntax
>>> HELLO_variable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'HELLO_variable' is not defined
```

- Variables can be reassigned arbitrarily often to change their value
- **Variables do not have specific types!**
  → **dynamic semantics**

- Variable names can consist only of letters, numbers, and underscores (_), and they can't start with numbers
- Python is **case sensitive**

del Variables

- Trying to reference a variable you have not assigned to causes an error
- Use del statement to remove a variable (i.e. reference from the name to the value is deleted); deleted variables can be reassigned later as usual

```
>>> hello = "test 1"
>>> hello
'test 1'
>>> world
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'world' is not defined
>>> del hello
>>> hello
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'hello' is not defined
```

`del x` **deletes the variable name**, not necessarily the actual data immediately.

the name “hello” disappears, the object “test 1” becomes unreferenced (no label attached to it). Whether the memory is freed instantly depends on Python’s garbage collector.

`del` removes the reference, not the memory itself.

What is Garbage Collection (GC) in Python?

Python uses **reference counting + garbage collector** to manage memory.

```
import gc
gc.collect()
```

1. scans for unreachable objects
2. removes cyclic references
3. frees memory immediately
4. returns number of objects collected

In normal Python programming → **rarely needed**.
But it's very useful in:

- GPU memory cleanup (PyTorch, TensorFlow)
- long-running Jupyter Notebooks
- machine learning training loops
- simulation code that creates many temporary objects
- cleaning memory before a big experiment

del removes only the name
gc.collect() removes the actual object(if nothing refers to it)

```
x = [1,2,3]
y = x
del x
print(y)  # still exists
```

Print

```
>>> print("Hello world!")
Hello world!
```

built-in function
For debugging or output display

Python has the capability of carrying out calculations

Standard operators, +, -, * and / work as you would expect; parenthesis () can be used for grouping

```
>>> 3 + 4
7
>>> -2 - 1 + 5
2
>>> (50 - 5*6) / 4
5.0
>>> 8 / 5
1.6
>>> 6 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

Will return a float

ZeroDivisionError, divisor (denominator) can't be zero

Output: The print() function

The **print() function** produces a more readable output, by omitting the enclosing quotes and by printing escaped and special characters

```
>>> d = "First line.\nSecond line."
>>> d
'First line.\nSecond line.'
>>> print(d)
First line.
Second line.
```

- **“d”** is a **variable** and allows you to store a value using *one equals sign*.

print format

- String Formatting with %: Older style of string formatting, useful for formatting numbers or strings into a template.

```
name = "Bob"
age = 25
print("%s is %d years old." % (name, age))
```

- String Formatting with .format(): More modern than % formatting, provides greater control and readability.

```
name = "Carol"
age = 30
print("{} is {} years old.".format(name, age))
```

Highly Recommend!!!

- Formatted String Literals (f-strings): Introduced in Python 3.6, this is the most concise and readable way to format strings.

```
name = "Dave"

age = 22

print(f"{name} is {age} years old.")
```

- Printing with Special Characters: Use escape sequences to include special characters in your strings, such as \n for a new line or \t for a tab.

```
print("First line.\nSecond line.\tIndented")
```

Using Python as a Calculator

```
>>> 17 // 4
4
>>> 17 % 4
1
>>> 5 ** 3
125
```

- Other useful operators:
  - // : floor division
  - % : remainder
  - ** : powers

```
>>> width = 5
>>> height = 3
>>> width * height
15
>>> width + _
20
>>> depth
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'depth' is not defined
```

- Assign a value to a variable using "="
- **Last printed expression assigned to the variable "_" (use as read-only variable)**

Using Python as a Calculator

- If you want to do more sophisticated calculations then *import* the *math* module.
- For example, take the square root of a number:

```
>>> 36**0.5
6.0
>>> import math
>>> math.sqrt(36)
6.0
```

Built-in numeric data types

- **int:** Integer numbers (e.g. 4, 6, -4)
- **float:** Decimal numbers (e.g. 5.0, -3.1)
- **complex:** Complex numbers; use j or J to indicate the imaginary part (e.g. 3+5j)

```
>>> (7+5j) + (8-3j)
(15+2j)
```

- **Operators with mixed type operands convert the integer operand to floating point**

Quiz time

1. Which option is the output of this code?

```
>>> (4 + 8) / 2
```

a) 6.0

b) 6

c) 8

Quiz time

1. Which option is the output of this code?

```
>>> (4 + 8) / 2
```

a) 6.0

b) 6

c) 8

Binary representation

>>> print(10 + 0.1 - 10) → **0.09999999999999964**

BUT !!!

Computers handle decimal numbers under the *IEEE 754 standard* for **floating-point arithmetic**

```
from decimal import Decimal

print(Decimal("10") + Decimal("0.1") - Decimal("10"))
```

✓ 0.0s

```
0.1
```

financial calculations, scientific computing, and other precision-sensitive domains

Strings

- Use a string if you want to use text in Python
- Strings can start/end with ‘ or “
- A string(result) is generally displayed using single quotes ‘

```
>>> "Python is amazing!"
'Python is amazing!'
>>> 'Lean back and enjoy this course'
'Lean back and enjoy this course'
```

Strings

What do you think will happen if we include “ in a string?

- Some characters cannot be directly included in a string
- Escape such special characters using a backslash before them

Escape	What it does.
\\	Backslash (\)
\'	Single-quote (')
\"	Double-quote (")
\a	ASCII bell (BEL)
\b	ASCII backspace (BS)
\f	ASCII formfeed (FF)
\n	ASCII linefeed (LF)
\N{name}	Character named name in the Unicode database (Unicode only)
\r	Carriage Return (CR)
\t	Horizontal Tab (TAB)
\uxxxx	Character with 16-bit hex value xxxx (u'' string only)
\Uxxxxxxxx	Character with 32-bit hex value xxxxxxxx (u'' string only)
\v	ASCII vertical tab (VT)
\ooo	Character with octal value ooo
\xhh	Character with hex value hh

Input: The input() function

The **input() function** prompts the user for input, and returns what they enter as a **string** (with contents automatically escaped).

```
>>> input("How are you? ")
How are you? this is what the user enters
'this is what the user enters'
>>> d = input("How old are you? ")
How old are you? 55
>>> d
'55'
```

- The input and print functions are very useful in programs **BUT NOT** at the Python console as input and output are done automatically.

String operations: **Concatenation** (1)

Concatenation is process of adding up strings in Python.

```
>>> "Spam" + 'eggs'
'Spameggs'
>>> print("First string" + ", " + "second string")
First string, second string
>>> "2" + "3"
'23'
>>> 2 + "3"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

- It does not matter if strings were created with “ or ‘
- Strings containing numbers are added up as strings and not integers
- Error when adding up string to a number (can be avoided with type conversion)

String operations: **Concatenation** (2)

```
>>> print("Hi!" * 3)
Hi!Hi!Hi!
>>> 2 * "3"
'33'
>>> "4" * "9"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't multiply sequence by non-int of type 'str'
>>> "Hi" * 3.0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't multiply sequence by non-int of type 'float'
```

- However, strings can be multiplied by integers (but not by other strings or floats)

String operations: **Concatenation** (3)

- Within the print()function, concatenation can be achieved via the "+" operator or via ","
- "," more convenient as it considers items to be concatenated independently from each other

>>> print("Hello "+"world!")

OR

>>> print("Hello ", "world!")

By default, print() with "," separates these items with a space:

```
print("Hello"+"world!")
print("Hello","world!")
```

✓ 0.0s

```
Helloworld!
Hello world!
```

Type conversion

- Some operations are impossible to complete due to type incompatibility

```
>>> "4" * "9"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't multiply sequence by non-int of type 'str'
>>> "4" + "9"
'49'
```

- Solution to this is **type conversion** (int(), float(), str())

```
>>> int("4") * int("9")
36
>>> int("4") + int("9")
13
```

Quiz time

1. What is the output of this code?

```
>>> hello = 3
>>> world = 2
>>> del hello
>>> world = 5
>>> hello = 4
>>> print(hello * world)
```

In-place operators

In-place operators allow you to write arithmetic operations, such as “x = x + 4”, more concisely as “x += 4”

```
>>> x = 5
>>> print(x)
5
>>> x += 3
>>> print(x)
8
```

Identical result

```
>>> x = x + 3
```

- Whatever is on the right hand side of the assignment “=“, gets computed first. Only after that, the result is assigned to the variable on the left hand side.
- In-line operations, e.g. x += 1 equivalent to x = x + 1. Python cannot do increment (++) or decrement (--) operators

In-place operators

In-place operators allow you to write arithmetic operations, such as “x = x + 4”, more concisely as “x += 4”

```
>>> x = 5
>>> print(x)
5
>>> x += 3
>>> print(x)
8
>>> x /= 4
>>> print(x)
2.0
```

- Also possible with other operators, such as -, /, *, %.

```
>>> var = "hello"
>>> print(var)
hello
>>> var += " world"
>>> print(var)
hello world
```

- Operators can also be applied to types other than numbers, e.g. strings

Commenting code properly

We receive this code...

```
1 user_name = input("Enter your name: ")
2 height = input("Enter your height in feet: ")
3 height = float(height)
4 height = height/0.032808
5 height = round(height,3)
6 height = str(height)
7 print(user_name + " is " + height + " cm tall")
```

Can you follow this code easily?

Commenting code properly

Then we receive this code…

- We can see:
  - Name of file and student
  - Comments on what lines do
- Goal: have 1 comment for every 1-4 lines of code
- Comment what code is doing and any design choices

```
4 #Obtain user name and height in feet
5 user_name = input("Enter your name: ")
6 height = input("Enter your height in feet: ")
7
8 #Convert feet into cm (1ft = 30.48cm)
9 #and round to 3 decimal places
10 height = round(float(height)*30.48,3)
11
12 print(user_name,"is",height,"cm tall")
```

Ten commandments of programming*

1. Think first about what you want to do before writing any code.
2. Break your problem into small independent blocks. Think about what precisely you want each block to do.
3. Prepare the project, i.e., create a folder and start with a fresh .py file.
4. Work "Inside to Outside": Identify the "core functionality" in each block, make it work first, and only then refine and extend.
5. If necessary, use the internet/ChatGPT to get help.
6. However, do not just copy-and-paste. **Understand the main idea** and write your own code!
7. Verification: Test your code line by line as you write it! Your computer always does exactly what you tell it to do. So if the result is different from the expectation defined in Step 2, it's your fault!
8. Read Python error messages. If you just added one line and the code "doesn't work" anymore, the problem is likely related to this line.
9. Always put meaningful comments into your code as you write it. Undocumented code is useless!
10. Be proud of what you achieved.

*Taken from http://nbviewer.jupyter.org/url/personalpages.manchester.ac.uk/staff/stefan.guettel/py/02a-loops_conditionals.ipynb#Ten-commandments-of-programming

Built-in types: Sequence types

- The principal built-in types are numeric, sequences, mappings, classes, instances and exceptions
- Sequence types are used to group together other values
- Three basic sequence types: **list, tuple, range**

Build-in types: Sequence types - list

- Lists are the most versatile sequence type
- Typically used to store collections of homogenous items (but items can also be of different types)
- Initialization of a list with n items:

  list_name = [item_0, item_1,…, item_n-1]

  (initialize an empty list using **list_name = list()** or **list_name = []**)
- Lists are **mutable**, i.e. items can be appended, removed, changed, etc after assignment.
- If you are going to iterate over the items then use a list (vs tuple, see later)

Build-in types: Sequence types - list

Accessing items in a list

```
>>> even = [2,4,6,8]
>>> even
[2, 4, 6, 8]
>>> even[0]
2
>>> even[-1]
8
>>> even[0:2]
[2, 4]
>>> even[2:4]
[6, 8]
>>> even[:2]
[2, 4]
>>> even[1:]
[4, 6, 8]
>>> even[-2:]
[6, 8]
```

- All built-in sequence types can be *indexed* and *sliced*
- While *indexing* is used to obtain **individual elements**, *slicing* allows you to obtain a
- Note how t… included and the end excluded
- Slice indices have useful defaults; an omitted first index defaults to zero, an omitted second index defaults to the size of the string being sliced.

Note: All slice operations return a new list containing the requested elements

Build-in types: Sequence types - list

```
 4 li =[2,4]
 5
 6 #Append an item with value 6
 7 li.append(6)
 8 print(li)
 9
10 #Append an item with value 6
11 li += [6]
12 print(li)
13
14 #Insert an item with value 99 at the 2nd position
15 li.insert(1,99)
16 print(li)
17
18 #Append items with value 8 and 10
19 li.append([8,10])
20 print(li)
21
22 #Append items with value 12 and 14
23 li.extend([12,14])
24 print(li)
```

Extending a list

```
[2, 4, 6]
[2, 4, 6, 6]
[2, 99, 4, 6, 6]
[2, 99, 4, 6, 6, [8, 10]]
[2, 99, 4, 6, 6, [8, 10], 12, 14]
```

Build-in types: Sequence types - list

```
5  li =[2,4,6,8]
6
7  #Change first item to 1
8  li[0] = 1
9  print(li)
10
11 #Change 2nd to 4th items to 2, 3, 4
12 li[1:] = [2,3,4]
13 print(li)
```

Replacing items in a list

```
[1, 4, 6, 8]
[1, 2, 3, 4]
```

Build-in types: Sequence types - list

```
 5 li =[2,4,6,8,10]
 6
 7 #Remove 2nd item in list
 8 del(li[1])
 9 print(li)
10
11 #Remove 2nd and 3th item in list
12 del(li[1:3])
13 print(li)
14
15 #Remove the item with value 10
16 li.remove(10)
17 print(li)
```

Removing items from a list

```
[2, 6, 8, 10]
[2, 10]
[2]
```

Build-in types: Sequence types - list

```
 4 li =[5,-2,3,1,7]
 5
 6 #Sort list in ascending order
 7 li.sort()
 8 print(li)
 9
10 #Sort list in descending order
11 li.sort(reverse=True)
12 print(li)
13
14 #Reverse list
15 li.reverse()
16 print(li)
```

Sorting a list

```
[-2, 1, 3, 5, 7]
[7, 5, 3, 1, -2]
[-2, 1, 3, 5, 7]
```

Build-in types: Sequence types - list

```
4 li =[1,5,-2,3,1,7]
5
6 #Check if a particular element is in the list
7 print(3 in li)
8 print(10 in li)
9
10 #Count the number of times an element is in the list
11 print(li.count(1), li.count(10))
12
13 #Obtain length of list
14 print(len(li))
15
16 #Retrieve min and max values of a list
17 print(min(li),max(li))
18
19 #Retrieve position of an item
20 print(li.index(5),li.index(1))
21 print(li.index(100))
```

Other useful operations for a list

```
True
False
2 0
6
-2 7
1 0
```

```
ValueError: 100 is not in list
```

Build-in types: Sequence types - tuple

- Tuples are **immutable**, i.e. items are fixed after assignment
- Used for collection of heterogeneous items, e.g. a person's details broken into (name, age, city)
- Generally used where order and position is meaningful and consistent
- Initialization of a tuple with n items:

  tuple_name = (item_0, item_1,…, item_n-1)

  (initialize an empty tuple using **tuple_name = ()** or **tuple_name = tuple()**)

Build-in types: Sequence types - tuple

Adding items to a tuple

```
1 tup = (1,2,3)
2 print(tup)
3
4 #use (x,) to add one element, x, to a tuple
5 to_add = (4,)
6 tup += to_add
7 print(tup)
8
9 #a list can be transformed into a tuple
10 to_add = [5,6,7,8]
11 ttt = tuple(to_add)
12 tup += ttt
13 print(tup)
```

```
(1, 2, 3)
(1, 2, 3, 4)
(1, 2, 3, 4, 5, 6, 7, 8)
```

lists

```
4 li =[2,4]
5
6 #Append an item with
7 li.append(6)
8 print(li)
9
10 #Append an item with
11 li += [6]
12 print(li)
13
14 #Insert an item with
15 li.insert(1,99)
16 print(li)
17
18 #Append items with v
19 li.append([8,10])
20 print(li)
21
22 #Append items with v
23 li.extend([12,14])
24 print(li)
```

Build-in types: Sequence types - tuple

Replacing items in a tuple

```
1 t = ('275', '54000', '0.0', '5000.0', '0.0')
2 print(t)
3
4 lst = list(t)
5 print(lst)
6
7 lst[0] = '300'
8 print(lst)
9
10 t = tuple(lst)
11 print(t)
```

```
('275', '54000', '0.0', '5000.0', '0.0')
['275', '54000', '0.0', '5000.0', '0.0']
['300', '54000', '0.0', '5000.0', '0.0']
('300', '54000', '0.0', '5000.0', '0.0')
```

lists

```
5 li =[2,4,6,8]
6
7 #Change first item t
8 li[0] = 1
9 print(li)
10
11 #Change 2nd to 4th i
12 li[1:] = [2,3,4]
13 print(li)
```

Build-in types: Sequence types - **range**

- The range type represents an **immutable** sequence of numbers
- The range type is commonly used for looping a specific number of times in for loops.
- Initialization a range object with a sequence of numbers starting at *start* and finishing at *stop* with a step size of *step:* **range_name = range(start, stop, step)**
- The advantage of the range type over a regular list or tuple is that a range object will always take the same (small) amount of memory, no matter the size of the range it represents.

Build-in types: Sequence types - range

```
4 #Create a range element from
5 #0 to 20 with step size 2
6 range_t = range(0,20,2)
7
8 #Print range variable
9 print(range_t)
10 print(list(range_t))
11
12 #Check if certain value is in range
13 print(10 in range_t)
14 print(11 in range_t)
15
16 #Obtain index of certain value
17 print(range_t.index(10))
18 print(range_t.index(11))
19
20 #Retrieve element(s) at a certain position
21 print(range_t[5])
22 print(range_t[:3])
23 print(range_t[-1])
```

```
range(0, 20, 2)
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
True
```

Yes

No

```
5
ValueError: 11 is not in range
```

```
10
range(0, 6, 2)
18
```

Build-in types: Sequence types - range

```
>>> list(range(0,10,1))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(0,30,5))
[0, 5, 10, 15, 20, 25]
>>> list(range(0,10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(0,10,3))
[0, 3, 6, 9]
>>> list(range(0,-10,-1))
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
>>> list(range(0))
[]
```

range_name = range(start, stop, step)
- default values are start = 0, step = 1

Build-in types: Text Sequence types - str

- Textual data in Python is handled with str objects, or strings
- Strings are **immutable** sequences of Unicode code points
- Initialization of a string (both ‘ and “ can be used):

  string_name = ‘text’ or **string_name = str(‘text’)**

- We can apply the same operations to strings as we do for other immutable sequence types, such as indexing, slicing, etc. (see next slide for examples)
- Strings have also many additional methods (overview of these can be found at https://docs.python.org/3/library/stdtypes.html#str)

Build-in types: Text Sequence types - str

```
>>> word = 'Python'
>>> word[0]
'P'
>>> word[-2]
'o'
>>> word[0:2]
'Py'
>>> word[0] = 'J'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> 'J'+word[1:]
'Jython'
>>> word[:2] + 'py'
'Pypy'
```

Build-in types: Text Sequence types - str

```
1 numbers = [1,2,3]
2 word = 'hello'
3 word_list = list(word)
4 numbers += word_list
5 print(numbers)
```

What is the output of this code?

Build-in types: Text Sequence types - str

```
1 numbers = [1,2,3]
2 word = 'hello'
3 word_list = list(word)
4 numbers += word_list
5 print(numbers)
```

What is the output of this code?

```
[1, 2, 3, 'h', 'e', 'l', 'l', 'o']
```

Why?

A string, let's call it *x*, is a **(text-)sequence**, i.e. if it is an input in list(x) or tuple(x), then the items in the list will be split.

Build-in types: Common sequence operations

- These operations are supported by most sequence types, both **mutable** (e.g. lists) and **immutable** (e.g. tuple)

Operation	Result
`x in s`	`True` if an item of *s* is equal to *x*, else `False`
`x not in s`	`False` if an item of *s* is equal to *x*, else `True`
`s + t`	the concatenation of *s* and *t*
`s * n` or `n * s`	equivalent to adding *s* to itself *n* times
`s[i]`	*i*th item of *s*, origin 0
`s[i:j]`	slice of *s* from *i* to *j*
`s[i:j:k]`	slice of *s* from *i* to *j* with step *k*
`len(s)`	length of *s*
`min(s)`	smallest item of *s*
`max(s)`	largest item of *s*
`s.index(x[, i[, j]])`	index of the first occurrence of *x* in *s* (at or after index *i* and before index *j*)
`s.count(x)`	total number of occurrences of *x* in *s*

Build-in types: **Mutable** sequence operations

Operation	Result
`s[i] = x`	item *i* of *s* is replaced by *x*
`s[i:j] = t`	slice of *s* from *i* to *j* is replaced by the contents of the iterable *t*
`del s[i:j]`	same as `s[i:j] = []`
`s[i:j:k] = t`	the elements of `s[i:j:k]` are replaced by those of *t*
`del s[i:j:k]`	removes the elements of `s[i:j:k]` from the list
`s.append(x)`	appends *x* to the end of the sequence (same as `s[len(s):len(s)] = [x]`)
`s.clear()`	removes all items from `s` (same as `del s[:]`)
`s.copy()`	creates a shallow copy of `s` (same as `s[:]`)
`s.extend(t)` or `s += t`	extends *s* with the contents of *t* (for the most part the same as `s[len(s):len(s)] = t`)
`s *= n`	updates *s* with its contents repeated *n* times
`s.insert(i, x)`	inserts *x* into *s* at the index given by *i* (same as `s[i:i] = [x]`)
`s.pop([i])`	retrieves the item at *i* and also removes it from *s*
`s.remove(x)`	remove the first item from *s* where `s[i] == x`
`s.reverse()`	reverses the items of *s* in place

Build-in types: **Mapping types - dict**

- Currently one standard mapping type in Python, the *dictionary*
- Dictionaries are indexed by *keys*, which can be any immutable type, e.g. strings, numbers or tuples.
- The *values* of a dictionary can be of any type.
- Think of a dictionary as an unordered set of *key: value* pairs, with the requirement that the keys are unique (within a dictionary).
- Initialization of a dictionary (more than two options):

dict_name = {key_1 : value_1,…, key_n : value_n} or **dict_name = dict([(key_1,value_1),…,(key_n,value_n)])**

Build-in types: **Mapping types - dict**

- Dictionaries have many useful operations, see https://docs.python.org/3/library/stdtypes.html#dict for an overview.

```
>>> tel = {'richard':1425, 'manuel':4769}
>>> tel
{'manuel': 4769, 'richard': 1425}
>>> tel['richard']
1425
>>> tel['anna'] = 9283
>>> tel
{'anna': 9283, 'manuel': 4769, 'richard': 1425}
>>> del tel['manuel']
>>> tel
{'anna': 9283, 'richard': 1425}
>>> 'anna' in tel
True
>>> tel.keys()
dict_keys(['anna', 'richard'])
>>> list(tel.keys())
['anna', 'richard']
```

The same dictionary can have key: value pairs of different types.

Build-in types: Mapping types - dict

```
>>> d = {"one": 1, "two": 2, "three": 3, "four": 4}
>>> d
{'one': 1, 'two': 2, 'three': 3, 'four': 4}
>>> list(d)
['one', 'two', 'three', 'four']
>>> list(d.values())
[1, 2, 3, 4]
>>> d["one"] = 42
>>> d
{'one': 42, 'two': 2, 'three': 3, 'four': 4}
>>> del d["two"]
>>> d["two"] = None
>>> d
{'one': 42, 'three': 3, 'four': 4, 'two': None}
```

```
>>> a = dict(one=1, two=2, three=3)
>>> b = {'one': 1, 'two': 2, 'three': 3}
>>> c = dict(zip(['one', 'two', 'three'], [1, 2, 3]))
>>> d = dict([('two', 2), ('one', 1), ('three', 3)])
>>> e = dict({'three': 3, 'one': 1, 'two': 2})
>>> a == b == c == d == e
True
```

Build-in types: Comparison

Lists: For ordered, mutable collections of items, especially when items might be added, removed, or changed, and the order of elements is significant.

Tuples: For fixed collections of items where immutability and integrity are necessary, or when using the collection as a key in a dictionary, or when memory efficiency is a concern.

Dictionaries: When working with associative arrays where you want to map unique keys to values for fast lookup, modification, and when the data is non-sequential.

13:00 – 14:00

Loops, tqdm, logging

Control flow statements - for

- The for statement is used to iterate over the elements of a sequence (e.g. a string, tuple or list) or other iterable object (typically *used if we know in advance* how may iterations should be done)

```
for  item in  sequence:
    indented statements to repeat; may use item
```

Control flow statements - for

Some more examples …

```
1 #A simple repeat loop
2
3 for i in range(10):
4     print('Hello')
```

```
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
```

```
6 n = int(input('Enter the number of times to repeat: '))
7 for i in range(n):
8     print('This is repetitious!')
```

```
Enter the number of times to repeat: 3
This is repetitious!
This is repetitious!
This is repetitious!
```

a = [1,2,3]

b = [2,3,5]

diff_a_b = [num for num in a if num is not in b]

Control flow statements - while

- The while statement is used for repeated execution as long as an expression is true
- Often used *if we do not know in advance* how may repetitions to be done)

```
while condition:
    indented statement block
```

```
for item in sequence:
    Indented statements to repeat; may use item
```

For VS While

for **item** in *sequence*:

    indented statements to repeat;
    may use **item**

```
1 n = int(input("n = "))
2 for i in range(n):
3     print(i)
```

For i in (0,1,2,3,…,n-1), print i

while *condition*:

    indented statement block

```
1 n = int(input("n = "))
2 i = 0
3 while i < n:
4     print(i)
5     i += 1
```

While i is less than n,

print i and increase it by one

Control flow statements - while

When is the condition checked?

```
i = 0
while i < 2:
    print("Before the increment:", i)
    i += 1
    print("After the increment:", i)
```

What is the output of this code?

```
Before the increment: 0
After the increment: 1
Before the increment: 1
After the increment: 2
```

while *condition*:

    intended statement block

Control flow statements - if

- The if statement is used for conditional execution

```
if condition:
    indented statement block if condition is true
else:
    indented statement block if condition is false
```

```
1 #Graduation example
2
3 credits = int(input("Input number of credits: "))
4
5 if credits > 120:
6     print("Congratulations! You are eligible for graduation!")
7 else:
8     remain = 120 - credits
9     print("Sorry, you need "+str(remain)+" credits more to graduate.")
```

Control flow statements - if

- Convert a numerical grade to a letter grade, ‘A’, ‘B’, ‘C’, ‘D’ or ‘F’, where the cutoffs for ‘A’, ‘B’, ‘C’, and ‘D’ are 90, 80, 70, and 60 respectively.

```
#Score example
score = int(input('What score do you have? '))
if score >= 90:
    letter = 'A'
else:   # grade must be B, C, D or F
    if score >= 80:
        letter = 'B'
    else:  # grade must be C, D or F
        if score >= 70:
            letter = 'C'
        else:    # grade must D or F
            if score >= 60:
                letter = 'D'
            else:
                letter = 'F'

print('Your grade is '+letter)
```

Control flow statements - if - elif

```
#Score example
score = int(input('What score do you have? '))
if score >= 90:
    letter = 'A'
else:   # grade must be B, C, D or F
    if score >= 80:
        letter = 'B'
    else:  # grade must be C, D or F
        if score >= 70:
            letter = 'C'
        else:    # grade must D or F
            if score >= 60:
                letter = 'D'
            else:
                letter = 'F'

print('Your grade is '+letter)
```

```
#Score example
score = int(input('What score do you have? '))
if score >= 90:
    letter = 'A'
elif score >= 80:
    letter = 'B'
elif score >= 70:
    letter = 'C'
elif score >= 60:
    letter = 'D'
else:
    letter = 'F'

print('Your grade is '+letter)
```

if *condition1*:

    indented statement block if condition1 is true

elif *condition2*:

    indented statement block if condition2 is true

elif *condition3*:

    indented statement block if condition3 is true

elif *condition4*:

    indented statement block if condition4 is true

else:

    indented statement block if each condition is false

Why do we need logging?

`print` is for:

- quick checks
- debugging during early development
- simple scripts or teaching

But it has **serious limitations** for real experiments, large projects, or production use.

Logging solves those limitations.

```
print("Training started")
```

```
import logging
logging.info("Training started")
```

Logging allows you to keep:

- time stamps
- verbosity levels
- warnings and errors
- debug messages
- experiment records
- persistent logs in files

logging

Level	Purpose
DEBUG	Developer details
INFO	Normal program flow
WARNING	Something unexpected, but still running
ERROR	Something failed
CRITICAL	Program might crash

```
import logging
logging.basicConfig(level=logging.INFO)

logging.info("Training started")
logging.warning("Learning rate is too high")
logging.error("Model crashed")
```

```
logging.basicConfig(level=logging.INFO)
logging.basicConfig(level=logging.ERROR)
logging.FileHandler("experiment.log")
```

logging – most used version

```
import logging

logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s - %(levelname)s -
%(message)s",
    handlers=[
        logging.FileHandler("experiment.log"),
        logging.StreamHandler()
    ]
)

logging.info("Experiment started")
```

→ save logs to `experiment.log`

→ print logs to terminal

What is tqdm?

`tqdm` is a **progress bar library** in Python.
Name comes from Arabic **“taqaddum” (progress)**.
It helps you visually track the progress of loops, tasks, and long-running operations.

from tqdm import tqdm

without tqdm, you see nothing in a loop, you may think:
“Is it running?”
“Is it stuck?”
“Should I stop it?”

```
for epoch in range(50):
    train_one_epoch()
```

VS

```
for epoch in tqdm(range(50)):
    train_one_epoch()
```

tqdm in ML, HPC, GPU training

When running jobs on:
- SLURM / PBS / LSF clusters
- remote GPU servers
- JupyterHub
- colab / notebook

`tqdm` makes it easy to track what your model is doing.

If progress suddenly slows:
DataLoader may be stuck
GPU/CPU may be overloaded
I/O bottleneck
memory leak

14:00 – 15:00

Functions, try and exception, OS

Functions

Motivation

- Python has a lot of built-in functions but what if we want to define our own functions?
- Code reuse beneficial especially for large projects → easier to maintain code (DRY vs WET principle)
- This is how a function call looks like...

```
function_name(function_arguments)
```

DRY: don't repeat yourself
WET: We enjoy typing or Write everything twice

Does this look familiar?

Functions

- Pre-defined functions vs **user-defined functions**
- Defining a function in maths, e.g. $f(x) = 5^x$
  - In Python this would be

`def` *function_name*(arguments):

indented statement block

```
10 def f_justPrint(x):
11     print("5 to the power of",x,"is",5**x)
12
13 f_justPrint(2)
```

```
5 to the power of 2 is 25
```

Functions

- You must define a function first before calling it
- Functions can have also zero arguments, e.g.

```
1 def printHelloWorld():
2     print("Hello world")
3
4 printHelloWorld()
```

```
Hello world
```

- or more than one argument (arguments can be of any type), e.g.

```
1 def print_sum_twice(x,y):
2     print(x + y)
3     print(x + y)
4
5 print_sum_twice(2,4)
```

```
6
6
```

Functions

- Function arguments and variables created inside a function cannot be referenced outside of the function's definition

`var` is a local variable

```
3 def suml(var):
4     var += 1
5     print(var)
6
7 suml(5)
8 print(var)
```

```
6
NameError: name 'var' is not defined
```

`var` is a local variable
`test` is a global variable

```
4 test = 4
5
6 def suml(var):
7     var += 1
8     print(var)
9     print(test)
10
11 suml(5)
```

```
6
4
```

Local vs global variables

- Python assumes variables are local, if not otherwise declared
- **Reason:** Global variables are generally bad practice and should be avoided. In most cases where you are tempted to use a global variable, it is better to utilize a parameter for getting a value into a function or return a value to get it out.
- So when you define variables inside a function definition, they are local to this function by default (even if the name is the same)

```
test = 4
def suml(var):
    var += 1
    test = 7
    print('is function, var', var)
    print('is function, test', test)

suml(5)
print(test)
```

```
in function, var 6
in function, test 7
4
```

```
3 def suml(var):
4     var += 1
5     print(var)
6
7 suml(5)
8 print(var)
```

```
6
NameError: name 'var' is not defined
```

`var` exists in the function definition block only

```
4 test = 4
5
6 def suml(var):
7     var += 1
8     print(var)
9     print(test)
10
11 suml(5)
```

```
6
4
```

As there is no local variable `test`, i.e. no assignment (=) to `test`, the value from the global variable `test` will be used

```
test = 4
def suml(var):
    var += 1
    test = 7
    print('is function, var', var)
    print('is function, test', test)

suml(5)
print(test)
```

```
in function, var 6
in function, test 7
4
```

There is an assignment to `test`, so this is a local variable in that block. The `test` variable outside the block is a global variable.

A variable (`test` in this case) can't be both local and global inside of a function.

```
 7 test = 4
 8 def sum1(var):
 9     print("in function, test",test)
10     var += 1
11     test = 7
12     print("in function, var",var)
13     print("in function, test",test)
14
15 sum1(5)
16 print(test)
```

```
  File "C:/Users/mbaxhra3/.spyder-py3/temp.py", line
9, in sum1
    print("in function, test",test)

UnboundLocalError: local variable 'test' referenced
before assignment
```

Use the keyword `global` to tell Python that you want to use a global variable

```
 7 test = 4
 8 def sum1(var):
 9     global test
10     print("in function, test",test)
11     var += 1
12     test = 7
13     print("in function, var",var)
14     print("in function, test",test)
15
16 sum1(5)
17 print(test)
```

```
in function, test 4
in function, var 6
in function, test 7
7
```

Functions

- `return` statement allows your function to return a value (otherwise it returns the special value `None`)
- Once a value from a function is returned, the function stops being executed immediately

```
4 def f_justPrint(x):
5     print("5 to the power of",x,"is",5**x)
6
7 f_justPrint(2)
```

```
5 to the power of 2 is 25
```

`def` *function_name*(arguments):

indented statement block

return argument

```
7 def f_justPrint(x):
8     print("5 to the power of",x,"is",5**x)
9
10 print(f_justPrint(2))
```

```
5 to the power of 2 is 25
None
```

Functions

- `return` statement allows your function to return a value (otherwise it returns the special value `None`)
- Once a value from a function is returned, the function stops being executed immediately

```
4 def f_justPrint(x):
5     print("5 to the power of",x,"is",5**x)
6
7 f_justPrint(2)
```

```
5 to the power of 2 is 25
```

`def` *function_name*(arguments):

Indented statement block

return argument

```
9  def f_withReturn(x):
10     return 5**x
11
12 y = f_withReturn(2)
13 print(y)
```

```
25
```

Functions

- `return` statement allows your function to return a value (otherwise it returns the special value `None`)
- Once a value from a function is returned, the function stops being executed immediately

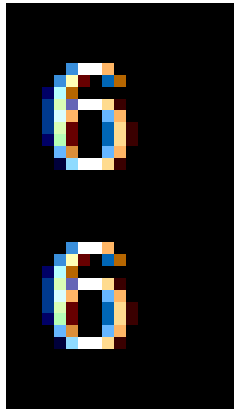
```
1  def min(x,y):
2      if x<=y:
3          return x
4      else:
5          return y
6      print("This would not be printed")
7
8  print(min(4,9))
9  z = min(3,1)
10 print(z)
```

For which values of x and y could this line be reached?

Functions as objects

- Although created differently from normal variables, functions are like any other kinds of value
- They can be assigned and re-assigned to variables

```
1 def multiply(x,y):
2     return x*y
3
4 x = 2
5 y = 3
6 product = multiply
7 print(product(x,y))
8 print(multiply(x,y))
```



Functions as objects

- Functions can also be used as arguments of other functions (Functional Programming)

```
1  def add(x,y):
2      return x+y
3
4  def do_twice(func,x,y):
5      return func(func(x,y),func(x,y))
6
7  a = 2
8  b = 3
9
10 print(do_twice(add,a,b))
```

Modules

- Pieces of code (`.py` files consisting of functions and values) that someone else has written to do a common task, e.g. mathematical operations

How to use a module?

- Add `import module_name` at the top of your code
- Use `module_name.var` to access functions and values with the name `var` in the module

```
1 import random
2
3 for i in range(5):
4     print(random.randint
```

Any idea what this code is doing?

Modules

- A second way of using modules in case you need **certain** functions from a module only:
  - Add "`from module_name import var`" at the top of your code
  - Use `var` as if it would be defined in your code
- Use comma separated list to import multiple objects
- `*` (e.g. `from math import*`) imports all objects from a module → Generally discouraged due to confusion of variables.

```
1 from random import randint
2 from math import pi, sqrt
3
4 for i in range(5):
5     print(randint(1,6))
6
7 print(pi)
8 print(sqrt(4))
9
10 pi = 4
11 print(pi)
```

Modules

- Import modules or objects under a different name using the `as` keyword
- Useful if a module or object has a long or confusing name

```
1 from math import pi as pi_value, sqrt as square_root
2
3 print(square_root(4))
4 print(pi_value)
```

Exceptions: Introduction

- Exceptions occur when something goes wrong due to incorrect code or input
- Without exception-handling, your program will terminate immediately in case of an exception

```
1 number1 = 7
2 number2 = 0
3 print(number1/number2)
```

```
>>> runfile('C:/Users/mbaxhra3/.spyder2-py3/ExceptionExample.py', wdir='C:/Us
pyder2-py3')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "C:\Users\mbaxhra3\AppData\Local\Continuum\Anaconda3\lib\site-packages
ets\externalshell\sitecustomize.py", line 699, in runfile
    execfile(filename, namespace)
  File "C:\Users\mbaxhra3\AppData\Local\Continuum\Anaconda3\lib\site-packages
ets\externalshell\sitecustomize.py", line 88, in execfile
    exec(compile(open(filename, 'rb').read(), filename, 'exec'), namespace)
  File "C:/Users/mbaxhra3/.spyder2-py3/ExceptionExample.py", line 3, in <modu
    print(number1/number2)
ZeroDivisionError: division by zero
```

Exceptions: Types

- Different exceptions are raised for different reasons
- Common exceptions include
  - **ZeroDivisionError:** division by zero
  - **ImportError:** an import fails
  - **IndexError:** indexing out-of-range item in a list
  - **NameError:** unknown variable is used
  - **SyntaxError:** code cannot be parsed properly (e.g. no “ **:** ” after if)
  - **TypeError:** operation or function is applied to an object of inappropriate type
  - **ValueError:** an operation or function receives an argument that has the right type but an inappropriate value

Quizz time

```
5  var_0 = 99
6  print(var_a)
7
8  print(var c)
9
10 del var_0
11 print(var_0)
12
13 1_var = 'hello'
14 print(1_var)
15
16 print("9" + 5)
17
18 print("Hello world\")
19
20 var_b = int("6.4")
21
22 list_a = [2,3,-3,2]
23 print(list_a[4])
24
25 if True and or False:
26     print("This is True")
```

NameError

SyntaxError

NameError

SyntaxError

TypeError

SyntaxError

ValueError

IndexError

SyntaxErrorr

- ZeroDivisionError
- ImportError
- IndexError
- NameError
- SyntaxError
- TypeError
- ValueError

5. What sort of errors would be raised in this code?

Exception handling

- To handle exceptions use `try` and `except` statements
- `try` statement contains code that may throw an exception, `except` statement defines what to do if a particular exception is thrown
- `try` statement can have more than one `except` clause. At most one `except` clause will be executed.

```
try:
```

```
try:
    number1 = 7
    number2 = 0
    print(number1/number2)
except (ZeroDivisionError,TypeError):
    print("An error ocurred due to zero division or incorrect variable type")
```

Use a single `except` clause with parentheses to capture multiple exceptions at once

Exception handling

- `except` statement without any exception specified will catch all errors
- Use these carefully… Any idea why?

```
1 try:
2     number1 = 7
3     number2 = 0
4     print(number1/number2)
5 except:
6     print("An error ocurred")
```

Exception handling

- Use a `finally` statement to ensure some code runs **regardless if an error occurs or not**, and what type the error is

```
1 try:
2     print("Hello")
3     number1 = 7
4     number2 = 0
5     print(number1/number2)
6 except ZeroDivisionError:
7     print("Divided by zero")
8 finally:
9     print("This code will run no matter what")
```

```
Hello
Divided by zero
This code will run no matter what
```

```
1 try:
2     print("Hello")
3     number1 = 7
4     number2 = 0
5     print(number1/number2)
6 finally:
7     print("This code will run no matter what")
```

```
Hello
This code will run no matter what
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "C:\Users\mbaxhra3\AppData\Local\Continuum\Anaconda3
ets\externalshell\sitecustomize.py", line 699, in runfile
    execfile(filename, namespace)
  File "C:\Users\mbaxhra3\AppData\Local\Continuum\Anaconda3
ets\externalshell\sitecustomize.py", line 88, in execfile
    exec(compile(open(filename, 'rb').read(), filename, 'ex
  File "C:/Users/mbaxhra3/.spyder2-py3/ExceptionExample.py"
    print(number1/number2)
ZeroDivisionError: division by zero
```

OS and pathlib

`os` and `pathlib` allow Python to interact with your computer's **file system**:

- read folders
- rename files
- create directories
- move / delete files
- check file size
- change working directory
- run system commands

For data science, ML, automation, and HPC, file handling is essential.

listing files

```
import os
files = os.listdir("data")
print(files)
```

```
from pathlib import Path
files = list(Path("data").iterdir())
print(files)
```

- ✓ object-oriented
- ✓ cleaner syntax
- ✓ works well across OS (Win/Mac/Linux)
- ✓ integrates well with Jupyter & HPC scripts

renaming all files using a dict (mapping)

```
mapping = {
    "001.jpg": "cat.jpg",
    "002.jpg": "dog.jpg",
    "003.jpg": "bird.jpg"
}
```

```
from pathlib import Path

folder = Path("images")

for old_name, new_name in mapping.items():
    (folder / old_name).rename(folder / new_name)
```

Creating a Folder (only if not exists + parents=True)

```
output_path = "output/data"
import os
os.makedirs(output_path, exist_ok=True)
```

```
from pathlib import Path

output_path = Path("output/data”)
output_path.mkdir(parents=True, exist_ok=True)
```

`parents=True` → create parent folders if needed
`exist_ok=True` → no error if folder already exists

Markdown Introduction - Titles and Headings

- Headings are created by adding one or more # symbols before your heading text.
- The number of # symbols indicates the level of the heading

```
# Title (H1)
## Major Heading (H2)
### Subheading (H3)
#### 4th Level Subheading (H4)
```

Markdown Introduction – Emphasis (Bold)

- Bold: Wrap text with two asterisks or underscores (** ** or __ __).

```
**This is bold text**
__This is also bold text__
```

Markdown Introduction - Emphasis (Italic)

- Italic: Wrap text with one asterisk or underscore (* * or _ _).

```
*This text is italicized*
_This text is also italicized_
```

Markdown Introduction - Lists (Unordered Lists)

- Use asterisks, plus signs, or hyphens interchangeably.

```
- Item 1
- Item 2
  - Subitem 2.1
  - Subitem 2.2

* List 1
  * List 1.1
```

Markdown Introduction - Change font colour

```
<span style="color:red">Red text</span>
<font color = 'green'>Green text</font>
<b style= 'color:blue'>Blue text</b>
```

Bold + Colour

Markdown Introduction - Links and Images (1)

- Links: [Link text](URL)

```
[TumOnline](https://campus.tum.de/tumonline/pl/ui/$ctx/webnav.ini?pUrl=visitenkarte.show_vcard%3FpPersonenId%3D31C1A4D0930EE2D8%26pPersonengruppe%3D3)
```

Markdown Introduction - Links and Images (2)

- Images: Similar to links but prefixed with an exclamation mark.
- ![Alt text](Image_URL)

```
![TUM Logo](https://scalings.eu/tum-logo/)
```

Markdown Introduction - Code Blocks & Syntax Highlight

- Use triple backticks or indent with four spaces for multiple lines of code.
- Optionally, specify the language for syntax highlighting.

```python
def hello_world():
    print("Hello, world!")
```

Markdown Introduction - Change line

- Two spaces at the end of the line.
-

```
This is the first line.<br>
This line appears directly below the first line.
```

Contact details

- E-mail: yanfei.shan@tum.de
- Allow 0-3 working days for a reply

(please include MGT001437W in Subject line)

- Room: 12 Alte Akademie
- Office Hours: Tuesday, 10:30 AM – 17:00 PM;
  Wednesday, 13:00 AM – 17:00 PM