Programming for psychologists

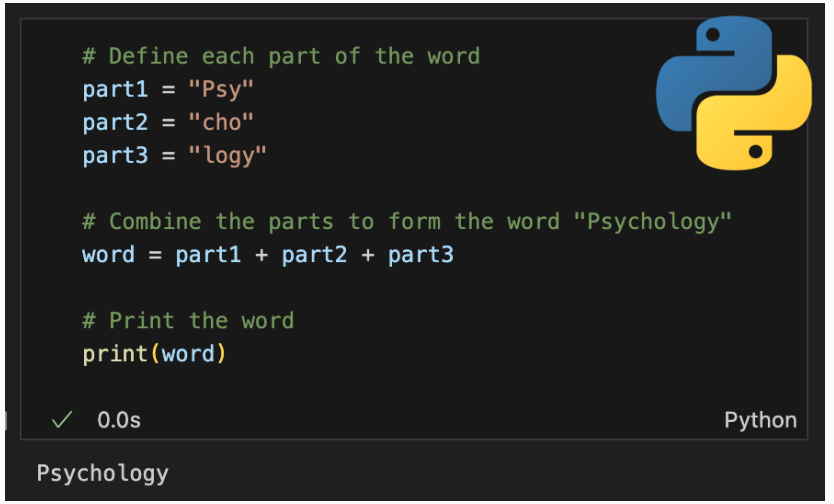
Lecture 2: Python

Matthias Nau

Python background

What is **Python** and why do we use it?

Python is a **powerful**, **general-purpose** programming language that is **widely used** in academia, medical research, and industry



Example code

Reasons for using Python include:

- Simple rules that are easy to learn
- It is popular within psychology and beyond
- Huge existing code base (e.g., many toolboxes)
- It has a strong online support community (e.g., *Stackoverflow.com*)
- Python is free for everyone (Open Science)

Why is **Python** popular in Psychology?

Python supports a wide range of essential tasks for psych/neuroscience research

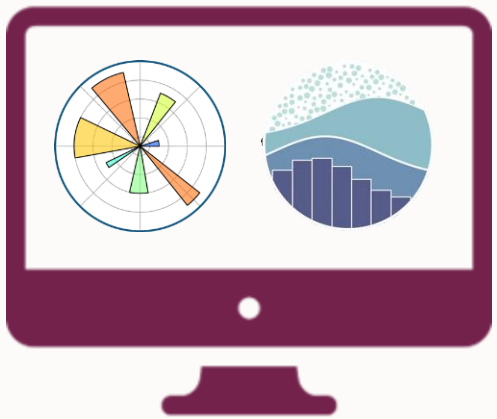
Experimental design & data acquisition

e.g., PsychoPy

Data analysis

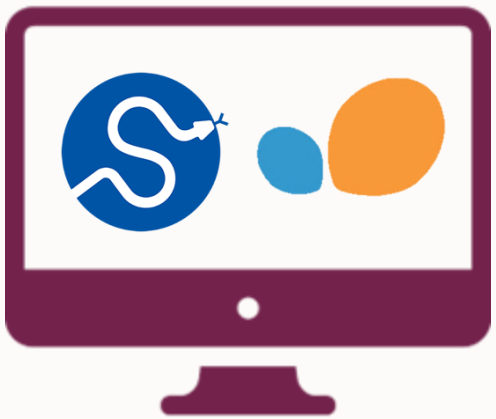
e.g., Numpy, Pandas

Visualization



e.g., Matplotlib, Searborn

Statistics & machine learning



e.g., Scipy, Sklearn

+ Many others

What if you do **NOT** want to continue in **Psychology**?

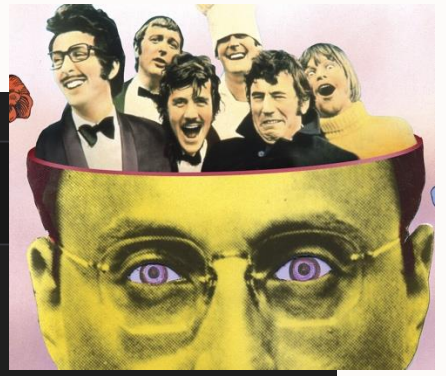
Great news! Your Python skills will be sought after in industry and beyond!

Who created **Python** when and where?

Guido van Rossum
Original creator 1989

Lead developer
until 2018

Centrum Wiskunde
& Informatica
Amsterdam



Named after
Monty Python

```
import this
```

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Zen of Python: Philosophy Easter egg

Who maintains **Python**?

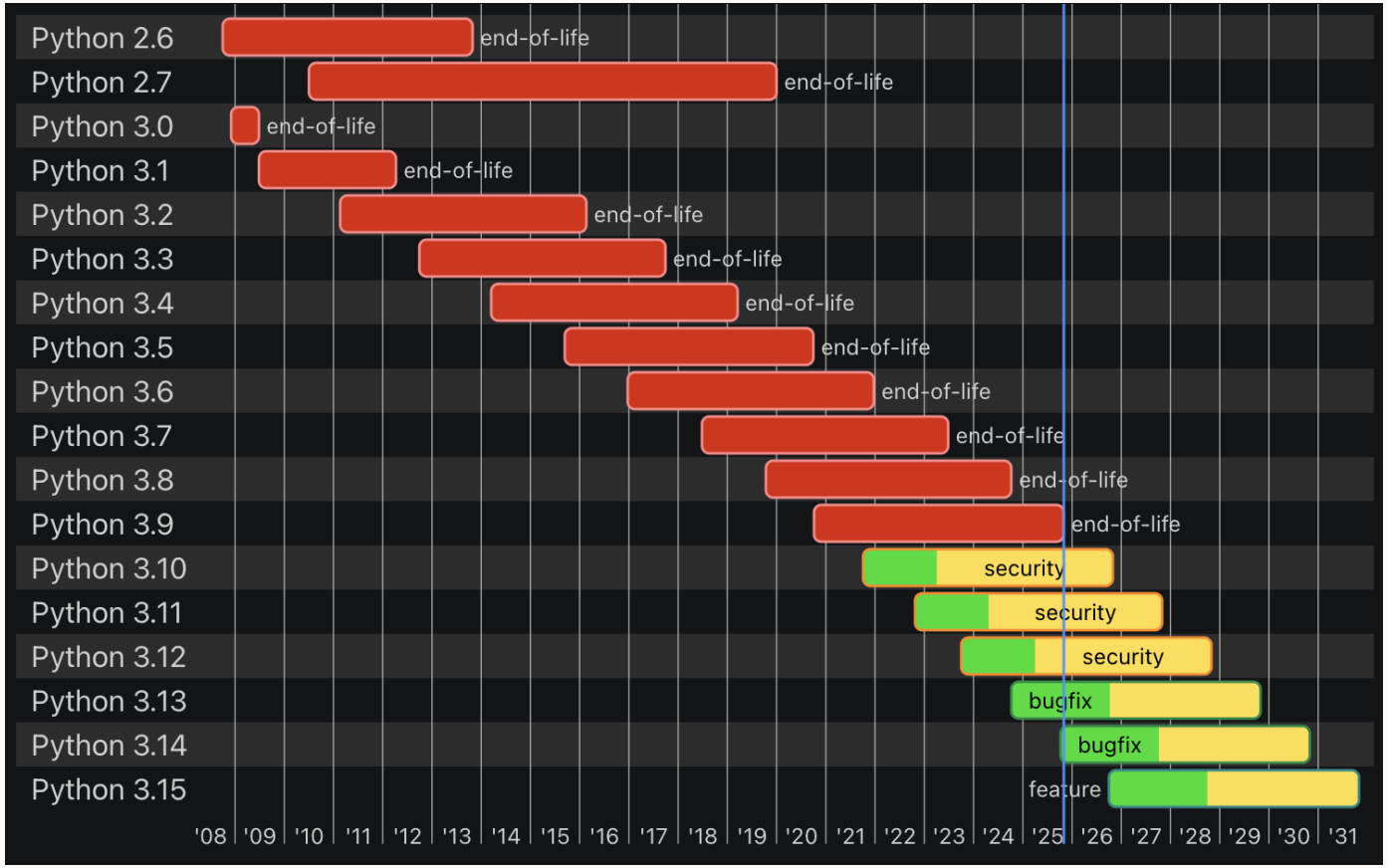
Python Software Foundation
Non-profit organization devoted to growing and enhancing Python.

Manages releases and legal issues, raises funds, and organizes conferences

Open-source community
There is a huge community sharing open code and tools in Python!

Check out: pyladies

Supporting maginalized genders to increase diversity among open-source community leaders



Many Python Versions (We will be using **Python 3.12.4**)

Python setup

(Recap practical 1.2)

Python Setup: Recap practical 1.2

VS Code

- Integrated development environment
- Main software running the show

Miniconda

- Creates and manages virtual environments
- Integrated into VS Code

Python

- Main programming language
- Included in Miniconda

Jupyter Notebooks

- Interactive documents for coding
- Included in Miniconda

Python Setup: Recap practical 1.2

Notebooks

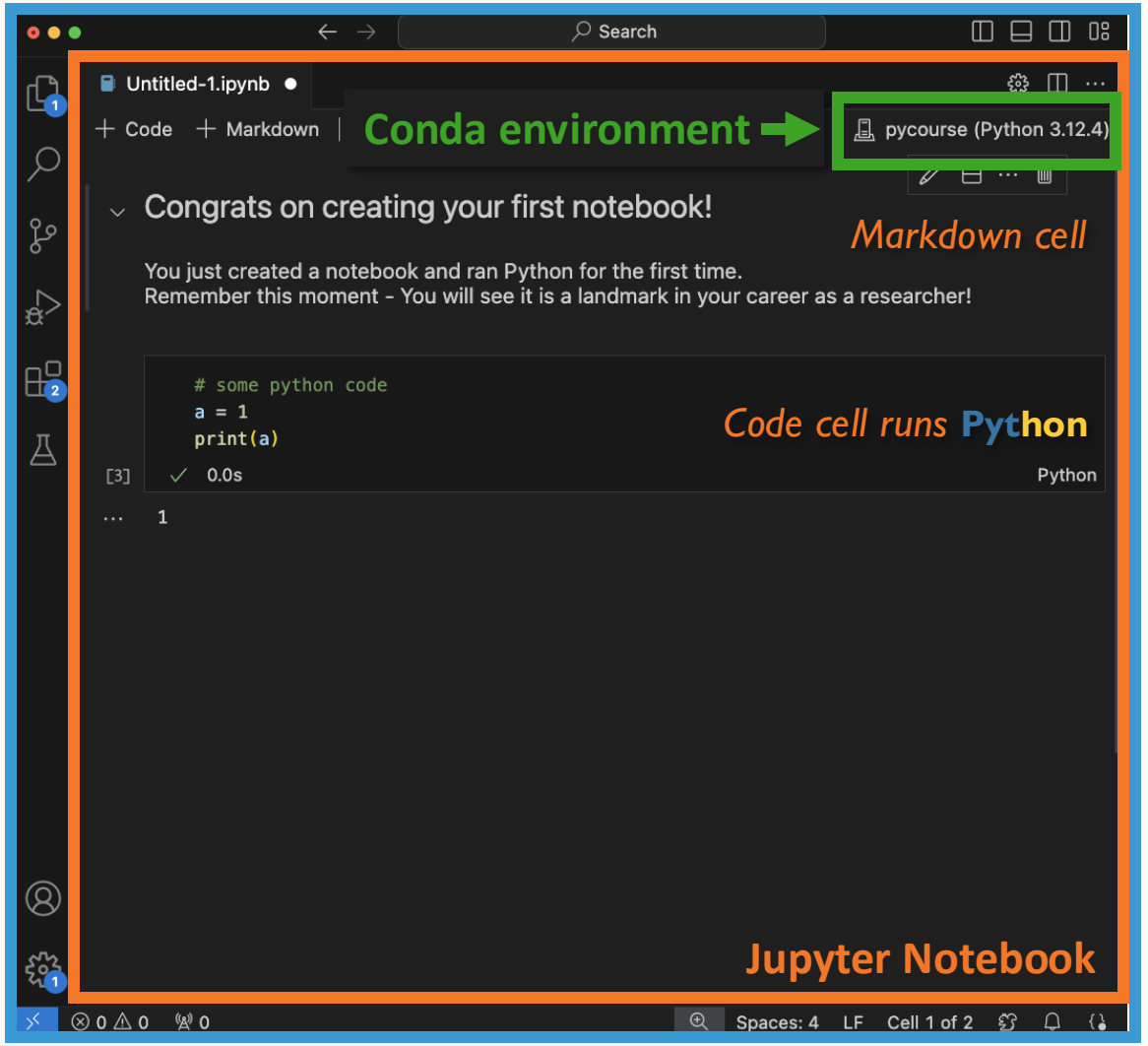
Remember, we use **VS Code** to run **Python** on a **Jupyter Notebook** inside a **Conda environment**.

Notebooks have two kinds of cells.

- **Markdown cells** for text and media.
- **Code cells** for writing and executing code.

Scripts

Most python code is written as scripts, text files that are then executed from the **Terminal** via the following command: **python script_name.py**.



Python Setup: Not part of practical 1.2

Python comes with a huge collection of pre-written code libraries called **Packages**.

Packages need to be installed with a **Package manager**.
The default package manager is called **PIP**.

Packages are installed through the **Terminal**

- Open terminal in **VS Code** (Click "**Terminal**" --> "**New Terminal**")
- Activate **Conda environment** (run "**conda activate pycourse**")
- Then install package (e.g., NumPy, run "**pip install numpy**")

Note that **conda** can install packages too ("**conda install numpy**")

Note:
Libraries need to be imported into your notebook or script before they can be used!

```
# import NumPy library
import numpy as np

# Define a NumPy array (datatype!)
my_array = np.array([1, 2, 3])

# Performing a simple operation
doubled_array = my_array * 2
```

Python basics

Python basics

Expressions are the most basic building blocks of programs: Short chunks of code that yield one or more **Items** when executed (e.g., numbers).

Expressions can be math equations (example 1-3), but are more general than that (example 4-6).

Expressions often include **Operators**, symbols that tell the computer to perform certain operations (e.g., + for addition).

Check out this overview on Python operators:
https://www.w3schools.com/python/python_operators.asp

Example expressions

```
# example 1
2 + 3

# example 2
2 * 3

# example 3
2 + 3 * 3

# example 4
3 == 3

# example 5
1

# example 6
print("Hello World!")
```

Example operators

==	Equal
!=	Not equals
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Python basics

We use the equal-sign operator to define **Variables**, labels linked to a certain item or value that can be referenced later in the code.

Calling variables (e.g., print(age)) causes Python to output their associated item or value (e.g., 25)

Operators allow combining variables to perform calculations, concatenate text etc. (e.g., c = a + b)

```
# Assigning values to variables
age = 25  # integers
height = 5.8  # floats
name = "Alice"  # strings
is_student = True  # booleans

# Using variables in calculations
next_year_age = age + 1
half_height = height / 2
```

Example variables

To use operators, Python needs to know the **Datatype** of each item and variable.

Python basics

There are **8 primary datatypes** in Python (many more in advanced code packages):

- **int:** Integer numbers without decimal point, example: 3
- **float:** Floating point numbers with decimal point, example: 3.1415
- **str:** Sequence of characters (text), example: "espresso"
- **bool:** Boolean value, example: True or False
- **tuple:** Ordered, immutable collection of values, example: (1, 2)
- **list:** Ordered, mutable collection of values, example: drink = ["espresso", "milk", "milk"]
- **set:** Unordered collection of unique values, example: drink = {"espresso", "milk"}
- **dict:** Dictionary, collection of key-value pairs, example: drink = {"espresso": 1, "milk": 2}

Quiz: Which of these examples are variables?

*mutability refers to the possiblity of manipulating values after they were defined, e.g., replacing a value in a list

Python basics

Accessing an item in a sequence (e.g, a list) is called **Indexing**.

The first element of a sequence has an **index of 0,** the second element has an index of 1 etc.

We can automatically index many parts of a sequence through **Loops**, which repeat blocks of code.

Loops are extremely useful for **repetetive tasks** (e.g., performing the same operation on data of 100 participants)

Indexing example

```
# define list
data = ['Alice', 'likes', 'Python']

# access second element
print(data[1])
```

✓ 0.0s Python

```
likes
```

For-loop example

```
# define for loop
for d in data:
    print(d)
```

✓ 0.0s Python

```
Alice
likes
Python
```

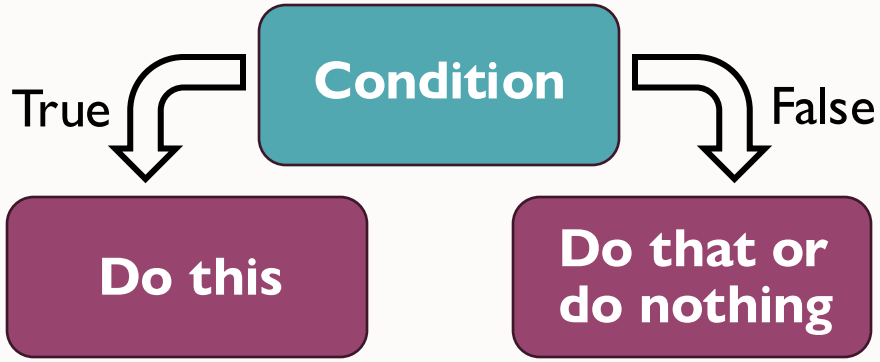
Python basics

Sometimes, your code needs to **do something depending on something else** (e.g., deciding which analysis to run depending on the data).

To steer what your code is doing, you can use **Conditionals** that allow making decisions.

The most common conditional is the **if-else statement** (see example), but there are others (e.g., match-case, assert)

Logic behind conditionals




```
# cinema visit
movie_recommended_age = 18
viewer_age = 20

# allowed to watch the movie or not?
if viewer_age >= movie_recommended_age:
    print("Here is your ticket. Have fun!")
else:
    print("Sorry, pick another movie!")
```

Python basics

An important Python skill is writing **Functions**, code blocks that perform a certain operation. (e.g., converting an **input** into an **output**).

Functions keep your code **organized**, **readable**, **easy to debug**, **re-usable**, and **easy to share.**

Functions are defined using the "**def**" function, and their inputs and outputs are called **Arguments**.

Function example

```python
# define a function
def my_function(input1, input2):
    output = input1 + ' likes ' + input2
    return output

# run the function & print output
output1 = my_function('Alice', 'Python')
print(output1)
```

✓ 0.0s Python

```
Alice likes Python
```

```python
# Reuse the same function & print output
output2 = my_function('Luke', 'Lightsabers')
print(output2)
```

✓ 0.0s Python

```
Luke likes Lightsabers
```

Python basics: Quiz!

Example 1

```
print(1 * 1)
print(2 * 2)
print(3 * 3)
print(4 * 4)
print(5 * 5)
print(6 * 6)
print(7 * 7)
print(8 * 8)
print(9 * 9)
print(10 * 10)
print(11 * 11)
print(12 * 12)
print(13 * 13)
print(14 * 14)
print(15 * 15)
print(16 * 16)
print(17 * 17)
print(18 * 18)
print(19 * 19)
print(20 * 20)
```

Example 2

```
# define function
def square(num):
    return num * num

# run function in a loop
for n in range(1,21):
    print(square(n))
```

Example 1 & 2 do the same thing!

Imagine you wanted to do this for thousands of numbers, which one would you choose? Why?

What about example 3?

```
# list of numbers
numbers = [1,2,3,4,5,6,7,8,9,10,11,12,
           13,14,15,16,17,18,19,20]

# define function
def square(num):
    return num * num

# loop over list & run function
for i in range(0,25):
    print(square(numbers[i]))
```

In principle ok,
BUT code crashes!

IndexError: list index out of range

Python basics

Python follows strict rules

If your code violates the rules, for example because the Datatype is wrong, Python will not execute it or crash.

Such terminations are called **Exceptions** or **Errors**, and they ensure that other programs are not affected.

Fixing errors and exceptions is called **Debugging**, which we will learn more about in Practical 2.2

https://docs.python.org/3/tutorial/errors.html

One solution: Try & except

This code runs through but tells you each time something is wrong

```
# list of numbers
numbers = [1,2,3,4,5,6,7,8,9,10,11,12,
           13,14,15,16,17,18,19,20]

# define function
def square(num):
    return num * num

# loop over list & run function
for i in range(0, 25):
    try:
        print(square(numbers[i]))
    except:
        print("something went wrong")
```

Python basics

Python follows strict rules

If your code violates the rules, for example because the Datatype is wrong, Python will not execute it or crash.

Such terminations are called **Exceptions** or **Errors**, and they ensure that other programs are not affected.

Fixing errors and exceptions is called **Debugging**, which we will learn more about in Practical 2.2

https://docs.python.org/3/tutorial/errors.html

Before the next practical, go through these slides again!

Do you know what the following terms mean?

- VS Code
- Conda environment
- Notebook
- Scripts
- Terminal
- Packages
- PIP
- Expressions
- Operators
- Variables
- Datatypes
- Indexing
- Loops
- Conditionals
- Functions
- Exceptions
- Debugging

Don’t forget your laptop on Monday!

Thanks - See you next week!