Programming for psychologists

Lecture 3: Python advanced

Matthias Nau

When you feel stuck…

When you feel stuck, remember…

- Remember that **this course is a safe zone**! You are not here to know, but to learn.
- **Separate code lines** visually to make it more manageable and less overwhelming (e.g., line breaks)
- Use the **print()** function to check values of variables.
- Use the **help()** function or **online documentation**.
- Use the **type()** function to check the data type (important for googling!).
- **Google your error message** (e.g.,"Python TypeError: 'int' object is not iterable").
- **Google your goal** (e.g., "How to remove duplicates from a list in Python").
- **Narrow down the problem** and go through the code **step by step** (e.g., comment out code and re-run the cell - does it still crash?).
- **Ask** fellow students or us for help
- Put on some **good music and try to relax!**

Functions

Functions

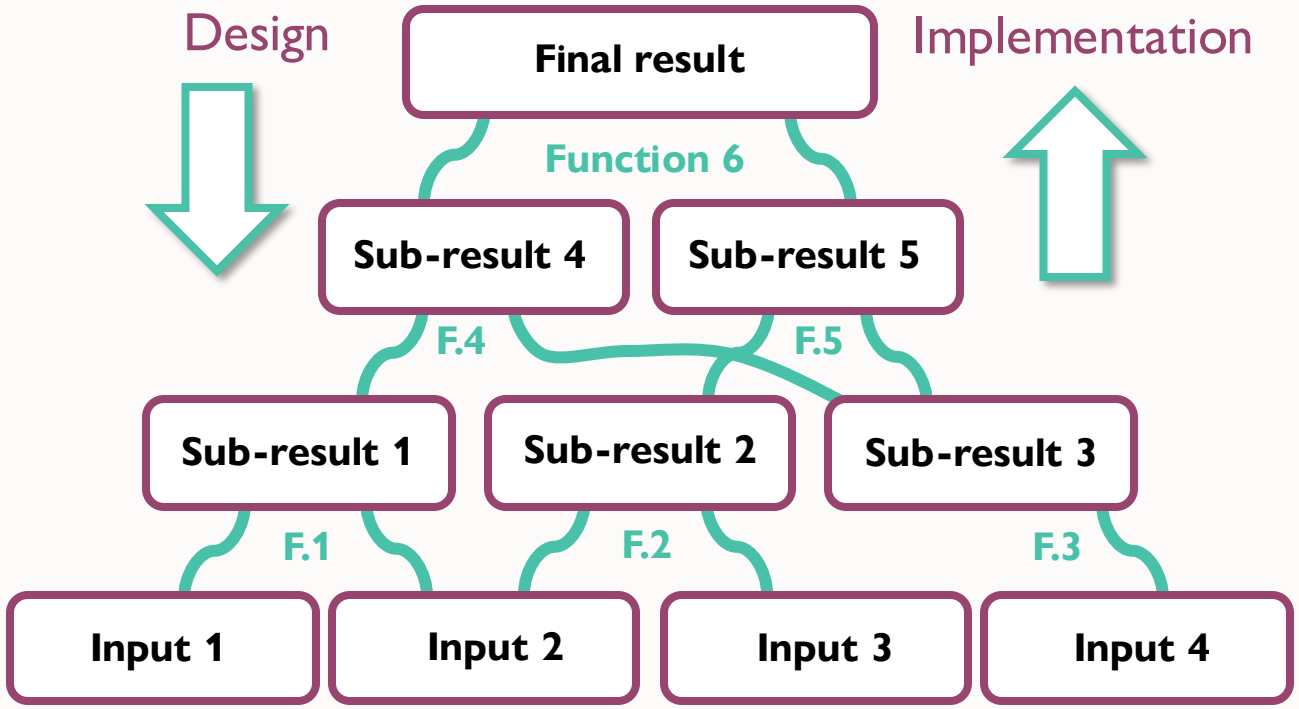
Functions are reusable blocks of code that perform specific tasks.

Why use functions?
They make your code…

- More readable and organized.
- More efficient (e.g., less copy-pasting).
- Easier to re-use.
- Easier to debug and maintain.
- Easier to design and implement.
- Easier to share.

Python has many great **built-in functions** (e.g., print()), but you can also write your own **user-defined functions**

Break down complex problems into simple parts



Functions – Basic syntax

Function are defined using the keyword **def,** followed by the function's **name** and **( )**.

Parameters are placed within the ( ).

The first statement, called the **docstring**, typically describes what the function does.

```
# Basic syntax of functions
def my_function(parameters):
    """docstring (optional): Describes the function."""

    # Here you write your code block
    value = sum(parameters) * parameters[-1] / parameters[1]

    return value
```

The docstring is followed by the **code block** you want to execute.

The statement **return** exits a function and passes back an expression to the caller (here the variable "value").

Functions – Basic syntax

Arguments are the values passed to a function when it is called.

Parameters are the variables the function expects based on the function's definition. Parameters can have default values.

Example function with default parameters

```
# function definition
def greet(message="Hello", name='Hannah'):
    my_str = f"{message} {name}"
    print(my_str)
```

Calling the same function with various input arguments

```
greet()
✓ 0.0s
```

```
greet(name="Clare")
✓ 0.0s
```

```
greet(message="Hey")
✓ 0.0s
```

```
greet(message='Hi', name="Sarah")
✓ 0.0s
```

Functions – Basic syntax

Return statements within the function pass a value back to the caller.
If there is no return statement, the function returns **None**

Function without return statement

```
# function definition
def greet(message="Hello", name='Hannah'):
    my_str = f"{message} {name}"


# call the function
X = greet()
print(X)
```

✓ 0.0s

Function with return statement

```
# function definition
def greet(message="Hello", name='Hannah'):
    my_str = f"{message} {name}"
    return my_str

# call the function
X = greet()
print(X)
```

✓ 0.0s

If your function should communicate anything back to the rest of the notebook, add a return statement!

Working with files

Working with files – Paths

As research psychologists, our experiments and analysis typically include **data files.**
The location of a file on your computer is called the **file path** of that file.

There are **two types** of file paths

- **Absolute file path**:
  The path to the file relative to the root directory, the hierarchically highest directory on your computer
  (e.g., /Users/matthiasnau/Desktop/Programming/Lecture1.mp4)

- **Relative file paths:**
  The path relative to the current working directory, typically initialized to where your notebook is saved.
  (e.g., Programming/Lecture1.mp4, relative to the desktop)

How to get the current working directory (cwd)

```
import os
os.getcwd()
```

✓ 0.0s

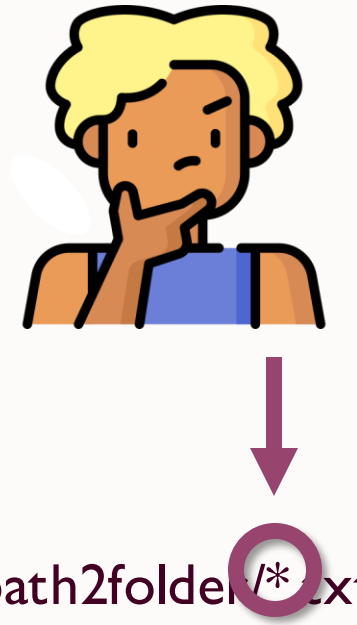
```
'/Users/matthiasnau/Desktop'
```

Working with files – Path managers

When reading or writing many files, copy-pasting file paths becomes highly impractical.
Luckily, Python comes with great functionalities to **find, read, and write files and folders**.

There are many **path management packages** in Python but my favorites are:

- **OS**
  - **os.getcwd()** Get current working directory
  - **os.path.join()** Joins paths correctly on Mac, Windows, Linux.
  - **os.listdir()** Lists all files and sub-directories in a directory.
  - **os.path.exists()** Checks if a file or directory exists.
  - **os.makedirs()** Create a new directory



- **Glob**
  - **glob.glob()** Finds all the file paths that match a pattern (e.g.,path2folder/*.txt).

Working with files - Wildcards

Often, we want to find all files of a specific type within a directory without knowing the exact file names.

In this case, we use a placeholder symbol, often called a "**wildcard character**". The symbol is *

You can also replace only parts of a file name with an asterisk to **search for more complex patterns**.

For example, glob.glob('*.ipynb') finds all notebooks, wheres glob.glob('M*.ipynb') finds all notebooks starting with the letter M.

```
# import libraries
import glob
import os

# where to search?
folder = '/path/to/your/directory'

# what file extension?
file_ex = '*.txt'

# create full path
find_files = os.path.join(folder, file_ex)

# find files
all_files = glob.glob(find_files)

# print the list of files
for file in all_files:
    print(file)
```

Working with files

Once we know the **location and names** of our files, we want to do something with them.
For example, we may want to **rename**, move, or copy them somewhere else to make **backups**.

My favorite package to do these things is called **Shutil** ("Shell utilities")

- **shutil.move()** Move and/or rename a file
- **shutil.copy()** Copy a file
- **shutil.copy2()** Copy a file incl. metadata
- **shutil.copytree()** Copy a directory
- **shutil.rmtree()** Delete a directory

Shutil can **save you a lot of time and nerves** by automating tasks that you do not want to do (e.g., renaming 1000 files manually)

Move and rename example

```
import shutil

# where is the file
source = 'folder1/my_file.docx'

# where should the renamed file go?
destination = 'folder2/my_file_renamed.docx'

# Move and rename the file
shutil.move(source, destination)
```

✓ 0.0s

```
'folder2/my_file_renamed.docx'
```

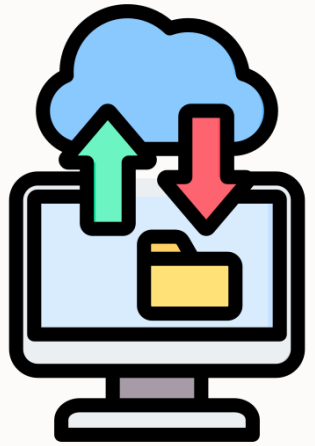
Working with files – General advice

- **Always have a backup of your raw data files**
  Before you process your data, make a backup

- **Avoid hardcoding paths, especially absolute file paths.**
  Instead, let a path manager do the work for you (e.g., OS).

- **If a file or folder can't be found, check the file path**
  Is it absolute or relative? On Mac, absolute file paths start with /
  On Windows, they start with a driver name (e.g., C:\) or \\

- **Avoid spaces in file or folder names**
  Many programs struggle with spaces. Instead of "my file.txt", call it "my_file.txt"



Consider using a cloud-based backup server

Style guide

Style guide

There are official coding conventions for Python summarized in a style guide called **PEP 8**.

Following PEP 8 improves your code's readability and robustness, and it supports code review.

Example I: How (not) to use white spaces

```
# Wrong:
spam( ham[ 1 ], { eggs: 2 } )
```

```
# Correct:
spam(ham[1], {eggs: 2})
```

```
# Wrong:
if x == 4 : print(x , y) ; x , y = y , x
```

```
# Correct:
if x == 4: print(x, y); x, y = y, x
```

Example II: How (not) to use line breaks

```
# Wrong:
# operators sit far away from their operands
income = (gross_wages +
          taxable_interest +
          (dividends - qualified_dividends) -
          ira_deduction -
          student_loan_interest)
```

```
# Correct:
# easy to match operators with operands
income = (gross_wages
          + taxable_interest
          + (dividends - qualified_dividends)
          - ira_deduction
          - student_loan_interest)
```

Style guide

Example III: How (not) to use indentations

```
# Wrong:

# Arguments on first line forbidden when not using vertical alignment.
foo = long_function_name(var_one, var_two,
    var_three, var_four)

# Further indentation required as indentation is not distinguishable.
def long_function_name(
    var_one, var_two, var_three,
    var_four):
    print(var_one)
```

```
# Correct:

# Aligned with opening delimiter.
foo = long_function_name(var_one, var_two,
                         var_three, var_four)

# Add 4 spaces (an extra level of indentation) to distinguish arguments from the rest.
def long_function_name(
        var_one, var_two, var_three,
        var_four):
    print(var_one)
```

Find the full style guide with many more examples here: https://peps.python.org/pep-0008/

Also, check out the VSCode extension: "*Black Formatter*" for automated formatting!

Home assignment

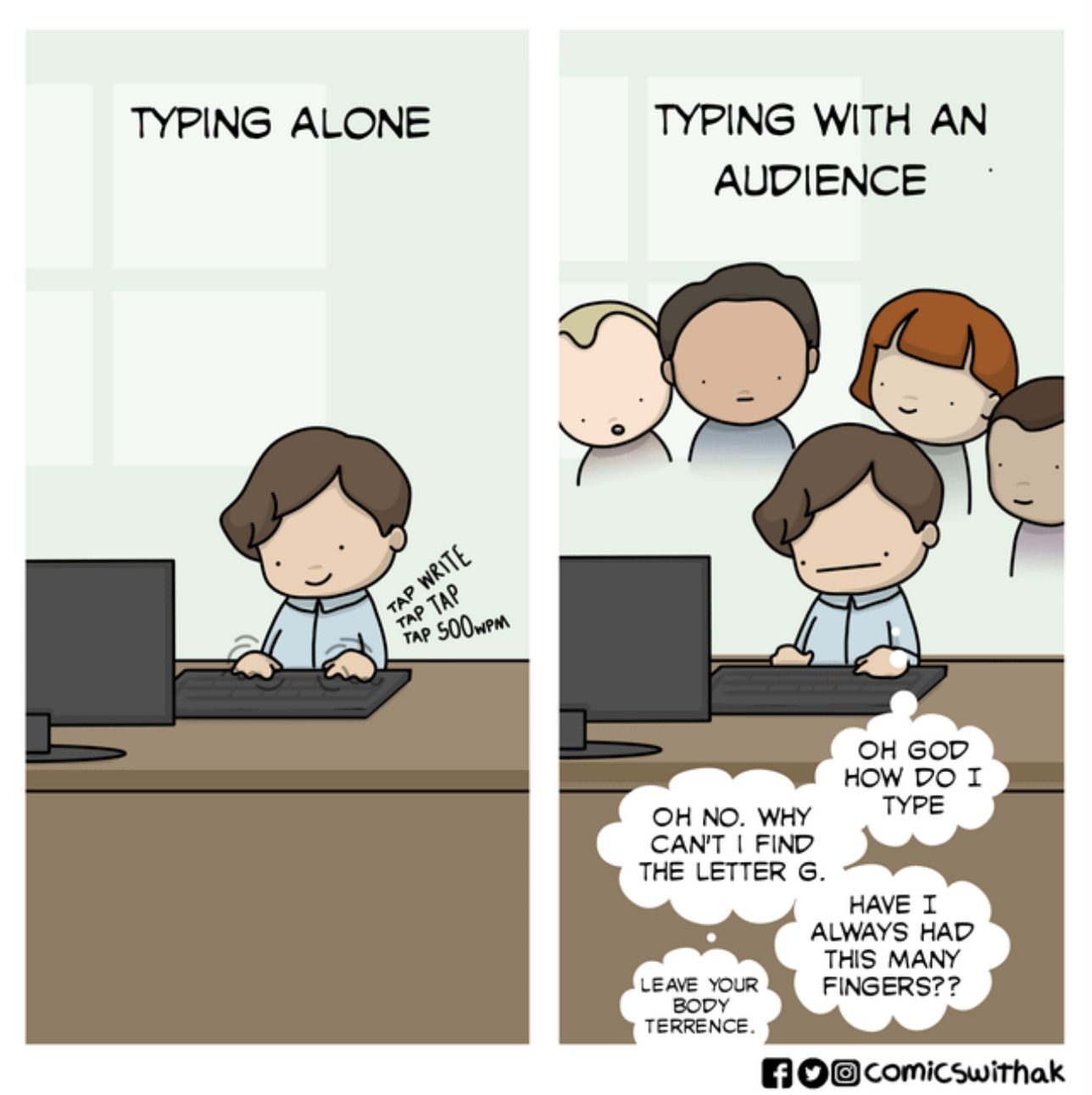
File Finder and Organizer in Python

- Write a Python function with three inputs:
  - A path to a **source folder** with files
  - A **file extension** (e.g., *.ppt).
  - A path to a **destination folder**
- The function should **search the source folder to find files** that match the file extension.
- The function should then **copy the files to the destination folder**.
  If the destination folder does not exist, the function should create it.

Show us your solutions in the practicals!

Optional

Coding at home
vs.
Coding in class



Before the next practical, go through these slides again!

Do you know what these statements mean?

- print(), type(), help()
- “There is no spoon”
- Style guide
- PEP8
- Function (built-in and user-defined)
- Parameter
- Argument
- Return statement
- Absolute file path
- Relative file path
- Path manager
- Wildcard character

That’s a wrap! See you in the practicals!