INF1-IP

Introduction to Programming

Lecture 11 – Modules

Matteo Degiacomi

THE UNIVERSITY of EDINBURGH
informatics

> Don't reinvent the wheel

Square roots, averages, random choices, file paths...
Someone has already written them, and tested them.
Python ships them as **modules**: the **standard library**.

Today:

- Three ways to import a module
- `dir()` and `help()` to explore unfamiliar tools
- File paths with `pathlib`
- Grouping our own functions into a module
- `main()` and `__name__`: definitions apart from demonstrations



Importing a module

`import` makes a module available. By convention, imports go at the **top** of the file.

Predict: which name is a number, and which are functions? What is printed?

```
import math

print(math.pi)
print(math.sqrt(81))
print(math.log(8, 2))
```

Output

```
3.141592653589793
9.0
3.0
```

- `math.pi` is a number; `math.sqrt()` and `math.log()` are functions
- Each module has its own **namespace**: `math.sqrt` is the name `sqrt` inside `math`

Two more import forms

Predict: what does each cell print? Which output changes when you run it again? Can `randint(1, 10)` give 10?

```
from random import choice, randint

random_nums = []
for i in range(1, 10):
    random_nums.append(randint(1, 10))

print(random_nums)
print(choice(random_nums))
```

```
[9, 9, 6, 3, 1, 2, 9, 8, 6]
8
```

```
import statistics as stats

values = [4, 7, 9, 10]
print(stats.mean(values))
print(stats.median(values))
```

```
7.5
8.0
```

Yes: `randint()` includes **both** ends, while `range(1, 10)` stops at 9.

> ~~Four~~ Three ways to import

Import	Then write	Good for
`import math`	`math.sqrt(81)`	most cases (clear function origin)
`from math import sqrt`	`sqrt(81)`	a few names, used often
`import statistics as stats`	`stats.mean(values)`	abridging long module names
`from math import *`	`sqrt(81)`	the brave (but generally avoid)

Avoid `from module import *`: it can silently replace your own names.

```
e = "maya@example.com"
from math import *
print(e)
```

→ `2.718281828459045`

`math` has a constant called `e`.

Who knew Maya's email was Euler's number?

{ Try it yourself #1: choose import forms

- Import `ceil` directly from `math`
- Import `statistics` as `stats`
- Round **up** every number in `[2.2, 3.8, 4.1]` with `ceil()`
- Print the mean of the rounded numbers

About 2 minutes · on your own

Example solution #1

```
from math import ceil
import statistics as stats

values = [2.2, 3.8, 4.1]
ceil_values = []

for val in values:
    ceil_values.append(ceil(val))

print(stats.mean(ceil_values))
```

→ 4

`ceil()` turns `[2.2, 3.8, 4.1]` into `[3, 4, 5]`.

Watch out! After you call `import statistics as stats`, the name is `stats`: `statistics.mean` would fail.

Inspecting a module: dir() and help()

```
import statistics

for name in dir(statistics):
    if "stdev" in name:
        print(name)
```

```
_mean_stdev
pstdev
stdev
```

```
import math

help(math.floor)
```

```
Help on built-in function floor in module math:

floor(x, /)
    Return the floor of x as an Integral.

    This is the largest integer <= x.
```

- `dir()` returns a **list** of names, so a loop can search it. Names starting with _ are conventionally treated as internal
- `help()` takes the function itself: `help(math.floor)`, not `help(math.floor(2.5))`

Try it yourself #2: investigate a module

- Find the names in `random` containing `sample` or `choice`
- Read `help(random.sample)`
- Pick two random items from `["A", "B", "C", "D", "E"]`
- What does the parameter `k` mean?

About 3 minutes · in pairs

Example solution #2

```
import random

letters = ["A", "B", "C", "D", "E"]

for name in dir(random):
    if "sample" in name or "choice" in name:
        print(name)

print(random.sample(letters, k=2))
```

```
choice
choices
sample
['D', 'A']
```

k is the number of items to pick.

`sample()` picks each position **at most once**, so no letter appears twice.

Working with paths: pathlib

Predict: here the `/` symbol does *not* mean division. What will each line print?

```
from pathlib import Path

relative_path = Path("ip26_l11_output") / "notes.txt"

print(relative_path)
print(relative_path.is_absolute())
print(relative_path.name)
print(relative_path.suffix)
print(relative_path.parent)
```

```
ip26_l11_output/notes.txt
False
notes.txt
.txt
ip26_l11_output
```

Watch out: two strings do not make a path: `"ip26_l11_output" / "notes.txt"` raises `TypeError`.

Few other useful methods:

- `Path.cwd()`: return the **c**urrent **w**orking **d**irectory
- `.resolve()`: return the absolute path
- `.exists()`: is there a file?

Live coding: folders and files

```
output_dir = Path("ip26_l11_output")
output_dir.mkdir(exist_ok=True)
message_path = output_dir / "message.txt"

with open(message_path, "w", encoding="utf-8") as file:
    file.write("Hello from a path object!\n")

print(message_path.name)
print(message_path.suffix)
print(message_path.parent)
```

```
message.txt
.txt
ip26_l11_output
```

- `.mkdir()` makes a folder.
- `exist_ok=True` argument: do not raise error if the folder exists
- `open()` accepts a `Path` directly
- `.name`, `.suffix`, `.parent`: parts of the path

Try it yourself #3: construct a path

- In the previous slide, we created `output_dir`
- Build a path to `reports/week1.txt` inside `output_dir`
- No need to create the file or the `reports` folder
- Print the path name, its parent, and its absolute form

About 2 minutes · on your own

Example solution #3

```
report_path = output_dir / "reports" / "week1.txt"
print(report_path.name)
print(report_path.parent)
print(report_path.resolve())
```

Output

```
week1.txt
ip26_l11_output/reports
...
```

The last line is the absolute path: *it is different on every computer.*

A function called main()

Move the **loose** code, outside any function, into `main()`. It is an ordinary function: it runs **only when called**.

Before

```
def total_cost(prices):
    return sum(prices)

basket = [2.4, 3.5, 2.8]
total = total_cost(basket)
print(f"Total: £{total:.2f}")
```

→ `Total: £8.70`

After

```
def total_cost(prices):
    return sum(prices)

def main():
    basket = [2.4, 3.5, 2.8]
    total = total_cost(basket)
    print(f"Total: £{total:.2f}")

main()
```

→ `Total: £8.70`

> Building our own modules

```
def convert_distances(distances_km):
    distances_miles = []
    for distance in distances_km:
        distances_miles.append(distance * 0.621371)
    return distances_miles

def format_distances(distances_miles):
    labels = []
    for distance in distances_miles:
        labels.append(f"{distance:.2f} miles")
    return labels

distances_km = [1, 5, 10]
converted = convert_distances(distances_km)
print(format_distances(converted))
```

→ `['0.62 miles', '3.11 miles', '6.21 miles']`

Module: a Python file containing functions, variables, classes, and other definitions that can be imported and used in another program.

separation of concerns: every module does **one job**, e.g.:

- `journey_tools.py`: convert
- `report_tools.py`: format

Module names: lowercase, with underscores. **Never** named as a module you use, e.g. `math.py` or `random.py`.

{ __name__ and the main guard

```
...

def main():
    distances_km = [1, 5, 10]
    converted = convert_distances(distances_km)
    print(format_distances(converted))

if __name__ == "__main__":
    main()
```

→ `['0.62 miles', '3.11 miles', '6.21 miles']`

The file is...	__name__
run directly	"__main__"
imported	its module name

Note: a notebook cell counts as *run directly*.

```
print(__name__)
```
→ `__main__`

{ Try it yourself #4: guard the demo

- Write `rectangle_area(width, height)`
- Write a `main()` that prints the area of a 4 by 7 rectangle
- Call `main()` inside a main guard

About 3 minutes · on your own

Example solution #4

```
def rectangle_area(width, height):
    return width * height

def main():
    print(rectangle_area(4, 7))

if __name__ == "__main__":
    main()
```

→ 28

In a notebook cell, `__name__` is `"__main__"`, so `main()` runs.

Imported from another file, the same code would print nothing.

Live coding: write a module

```
module_source = """def kilometres_to_miles(distance):
    return distance * 0.621371

def main():
    print("Running the journey demonstration")
    print(kilometres_to_miles(5))

if __name__ == "__main__":
    main()
"""

with open("ip26_l11_journey_tools.py", "w",
          encoding="utf-8") as file:
    file.write(module_source)
```

A whole program, kept in a **string**.

Write the string to a `.py` file (as in Lecture 10).

The new file is a **module**: **its name is the file name, without `.py`.**

Importing our own module

Predict: does the demonstration run? What is the module's `__name__`?

```
import ip26_l11_journey_tools
from ip26_l11_journey_tools import kilometres_to_miles

print(ip26_l11_journey_tools.__name__)
print(kilometres_to_miles(10))
```

```
ip26_l11_journey_tools
6.21371
```

→ inside the import, the guard is `False`.

Watch out: a module is loaded only on its **first** import. Changed the file? Restart the kernel.

Note: the new `__pycache__` folder speeds up the next import. You can ignore it, or delete it.

> Summary

- `import m`, `from m import name`, `import m as alias`; avoid `import *`
- `dir()` lists the names in a module, `help()` explains them
- `Path("folder") / "file.txt"` builds a path: `.name`, `.parent`, `.resolve()`
- One job per module: any `.py` file can be imported by its name
- `main()` holds the demonstration; `if __name__ == "__main__":` runs it only when the file is run directly
- Changed a module? Restart the kernel

Homework: the consolidation challenge at the end of the notebook: a temperature module with a main guard.