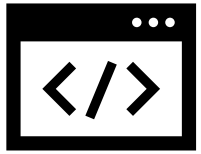
IMPERIAL



BIOE50010 – Programming 2

Computer Lab 3: Modular Programming

Binghuan Li, Maria Portela, Gauthier Boeshertz, Samuel George-White, Yilin Sun, Kamrul Hasan, Wenhao Ding, Siyu Mu, Lito Chatzidavari

20 October, 2025

Feedback on Week 2 - **Paths**

Relative path and absolute path

- Using **absolute path**: starting from the *root directory*

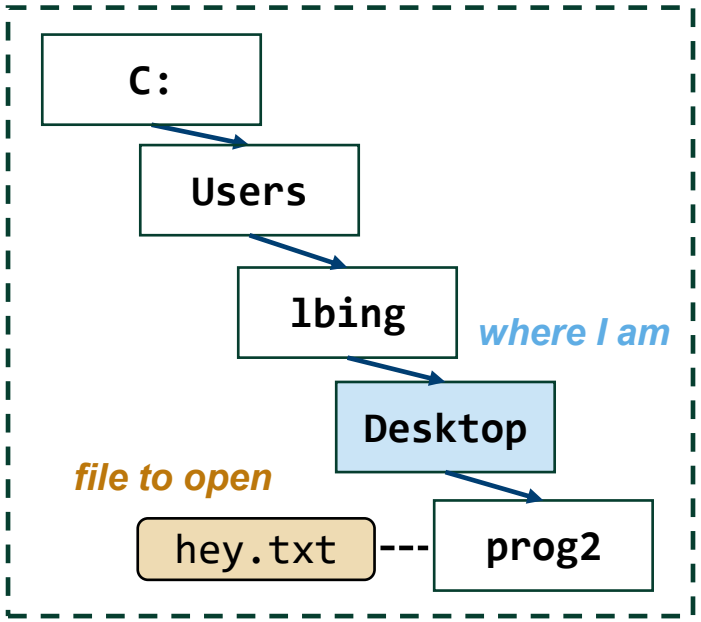
```
f = open('C:/Users/lbing/Desktop/prog2/hey.txt', 'w')
```

full path starting from `C:` disk

- Using **relative path**: with respect to the *current directory*

```
f = open('./prog2/hey.txt', 'w')
```

the current directory (…\Desktop)



- In relative path:
  - A single dot (`.`) refers to the current directory.
  - Double dots (`..`) refer to the parent directory (w.r.t. the current directory).
- **Caveat!** `./` (current directory) is different from `/` (root directory): the dot matters!

Feedback on Week 2 - **Formatting**

- **f**-string formatting starts with an **f** before the opening quotation mark.
- Each individual variable is enclosed within a pair curly brackets **{}**:

Code snippet from `extract_dna2protein.py`

```
TTT  Phe  F Phenylalanine
```

```
data = [['T', 'T', 'T', 'Phe', 'F', 'Phenylalanine'],
        ['T', 'T', 'C', 'Phe', 'F', 'Phenylalanine']]

print(f'{data[0][0]}{data[0][1]}{data[0][2]}\t{data[0][3]}\t{data[0][4]}\t{dat
a[0][5]}')
```

- Alternatively, one can use the **`format`** method:

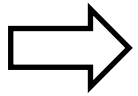
```
print('{}{}{}\t{}\t{}\t{}'.format(data[0][0], data[0][1], data[0][2],
data[0][3], data[0][4], data[0][5])
```

Feedback on Week 2 - **Raw String**

- By default, Python treats the backslash (\) as a special character: *e.g.*, `\t`, `\n`

Example

```
myStr = 'Hi\nHello'
print(myStr)
```



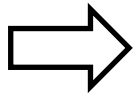
Console

```
>> Hi
Hello
```

- Python **raw string** ('**r**') treats the backslash as a literal character.

Example

```
myStr = r'Hi\nHello'
print(myStr)
```



Console

```
>> Hi\nHello
```

- …which can be useful to deal with the path separator (in Windows):

☹

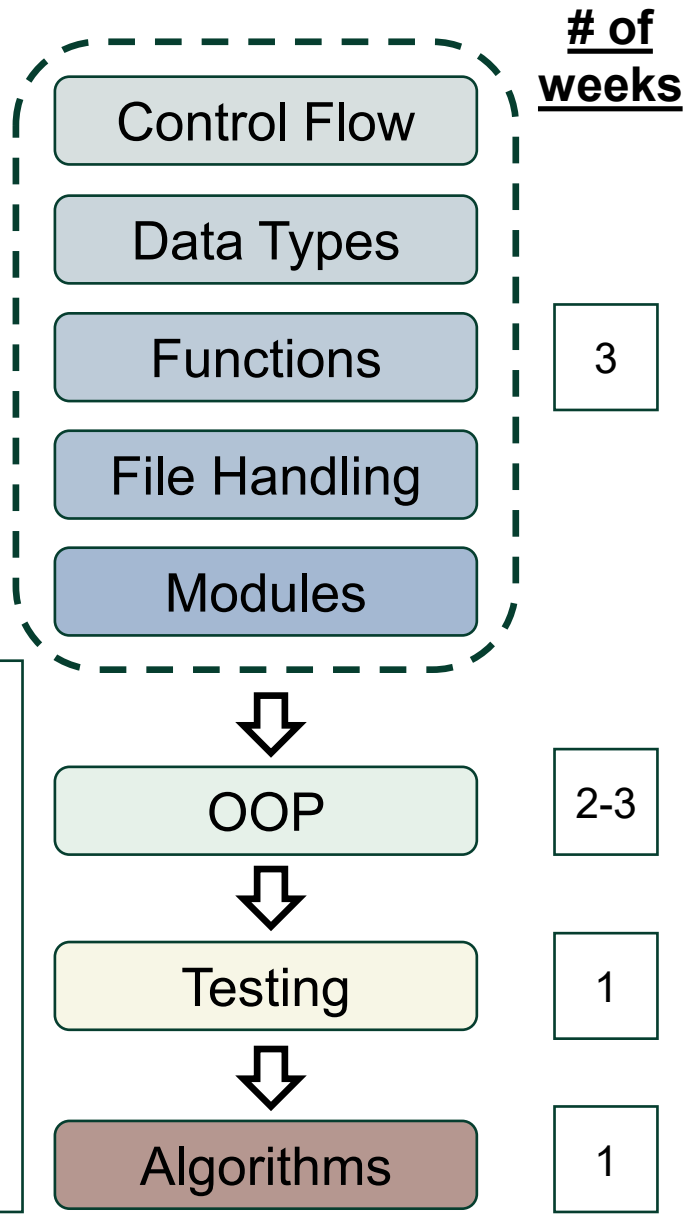
```
f = open('C:\Users\lbing\Desktop\lab2\the_road_not_taken.txt', 'r')
```

☺

```
f = open(r'C:\Users\lbing\Desktop\lab2\the_road_not_taken.txt', 'r')
```

Progress Check

Week 3:
we are here



Revision Points (from week 2)

- **File I/O**: open, read, write, close.
- Using loops to read lines recursively.
- Print formatting with **f-string**
- **String methods**: `.count()`, `.strip()`, `.split()`
- **List methods**: `.append()`

Your tasks today

- Four mini tasks on **modular programming**:
  - Task 1: Calculate radius and angle from a pair of Cartesian coordinates

$$\theta = \left(\text{atan}\left(\frac{y}{x}\right)/\pi\right) * 180°$$

  - Task 2: Passing an unknown number of arguments into a function
  - Task 3: The Collatz conjecture
  - Task 4: Plotting marks on a user-defined board

To start…

- Read all information and the **sample output** provided in the lab carefully
- Consult the help pages for the string / list methods provided in *Lab 2 slides*
- Study the non-keyword and keyword arguments attached to the slides.

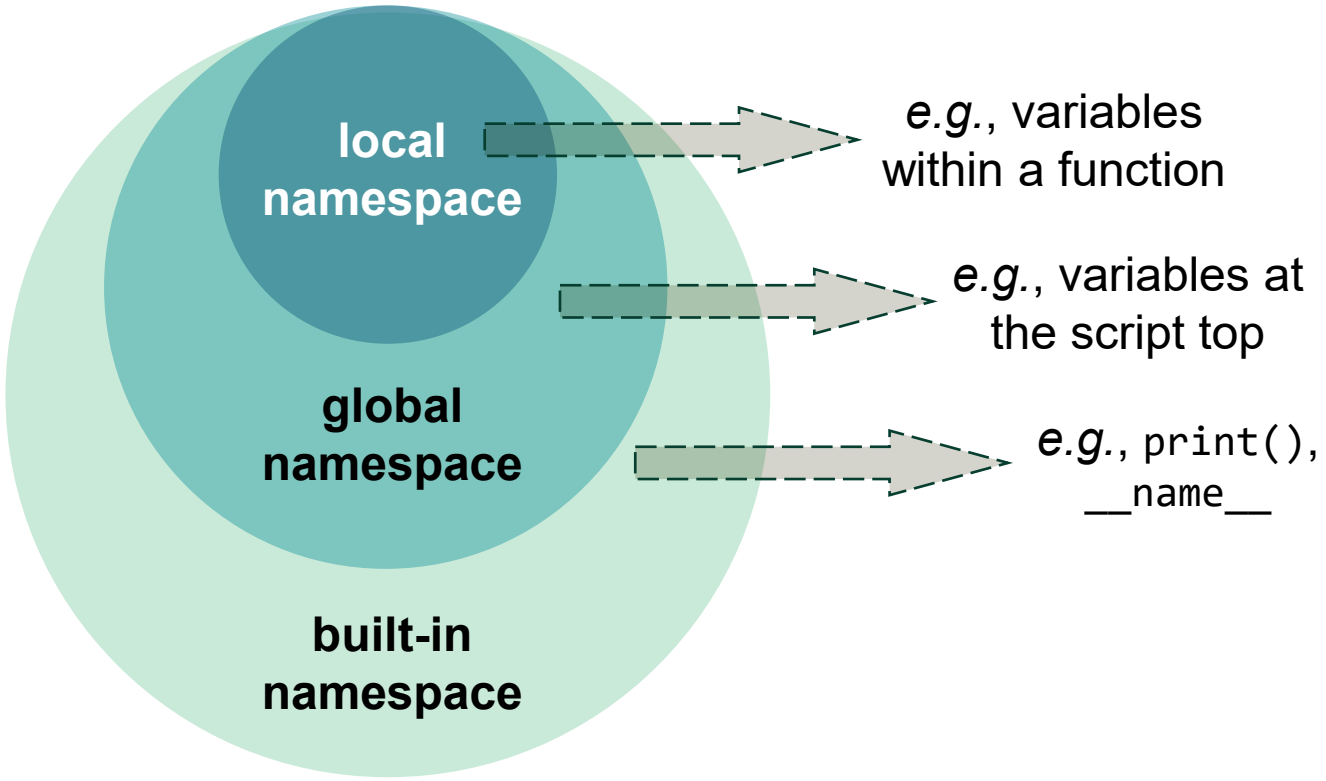
Questions?

That's it for now.

You can now proceed to the Lab 3 exercises.

Appendix 1: **Namespace**

- A **namespace** holds a set of names that belongs to a specific context (**scope**) within the program.
- If you create a variable within a function, that variable *only* exists in that function.



Example

```
x = 'global!'

def print_x():
    x = 'local!'
    print(x)

def main():
    print_x()
    print(x)

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

same variable name but hold **different** values

Console

```
>> global!
Local!
```

Appendix 2: **Non-Keyword Argument Functions**

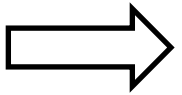
Suppose you are defining a function with arbitrary number of arguments…

- You can use non-keyword argument functions (*arg)

Example

```
def good_fruits(*fruits):
    for item in fruits:
        print('let us take a', item)

good_fruits('kiwi', 'watermelon', 'durian')
```



Console

```
let us take a kiwi
let us take a watermelon
let us take a durian
```

Comments

- The asterisk * is known as the unpacking operator.
- All *args are collected and packed into a **tuple** (hence, use loops)
- Positional arguments must come before *args:

positional argument

*args

```
def call_greeting(greeting, *names)
```

Appendix 3: **Keyword Argument Functions**

- Alternatively, you can use keyword argument functions (**kwarg)

Example

```
def good_fruits(**fruits):
    for fruit, attribute in fruits.items():
        print(f"Let us take a {fruit}, which is {attribute}.")

good_fruits(kiwi="green", watermelon="large", durian="spiky")
```

"keywords"

Console

```
Let us take a kiwi, which is green.
Let us take a watermelon, which is large.
Let us take a durian, which is spiky.
```

Comments

- All *kwargs are collected and packed into a **dictionary** ({key}:{value})
- Positional arguments and *args must come before **kwargs.

Appendix 4: `enumerate()` and `range()`

These are two useful functions to iterate over sequences in loops.

- `range(start=0, stop, step=1)` - iterate through a sequence of numbers
- `enumerate(iterable, start=0)` - iterate through an *iterable object* (list, tuple, dictionary) and keep track of the index.

Example

```
good_friuts = ['kiwi', 'watermelon', 'durian']

# using range()
for idx in range(0, len(good_friuts)):
    print(f'{idx} {good_friuts[idx]}')

# using enumerate()
for idx, fruit in enumerate(good_friuts):
    print(f'{idx} {fruit}')
```

same output!

Console

```
0 kiwi
1 watermelon
2 durian
```