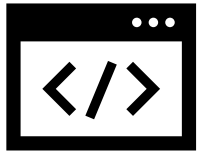
IMPERIAL



BIOE50010 – Programming 2

Computer Lab 8

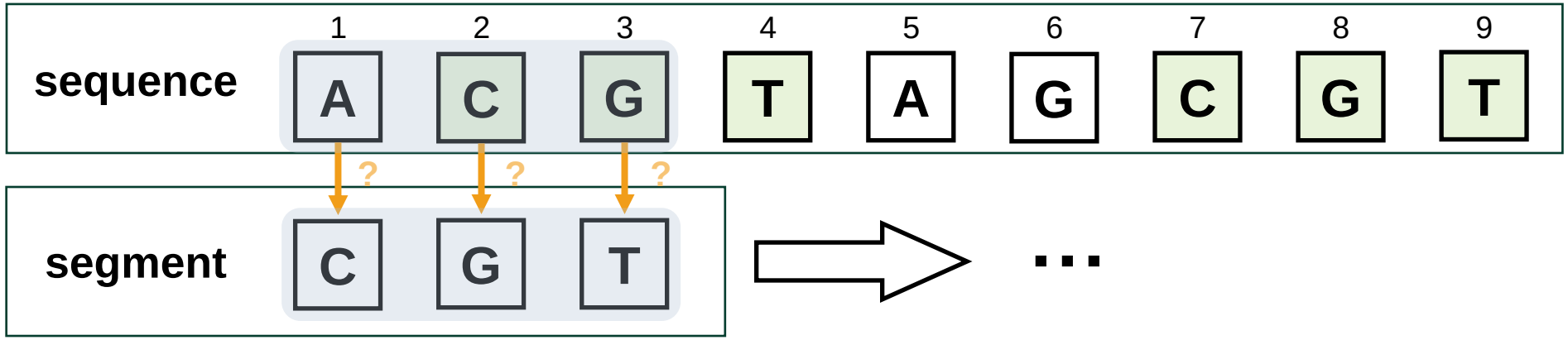
Binghuan Li Department of Chemical Engineering

Maria Portela Department of Bioengineering

Wenhao Ding Department of Bioengineering

24 November, 2024

`find()`



- Two ways to realise the matching algorithm: **`find()`** and **`find_faster()`**
  - **Element to element** comparison: intuitive to code, but slow (2 `for` loops)
  - **List to list** comparison: concise and faster (1 `for` loop)

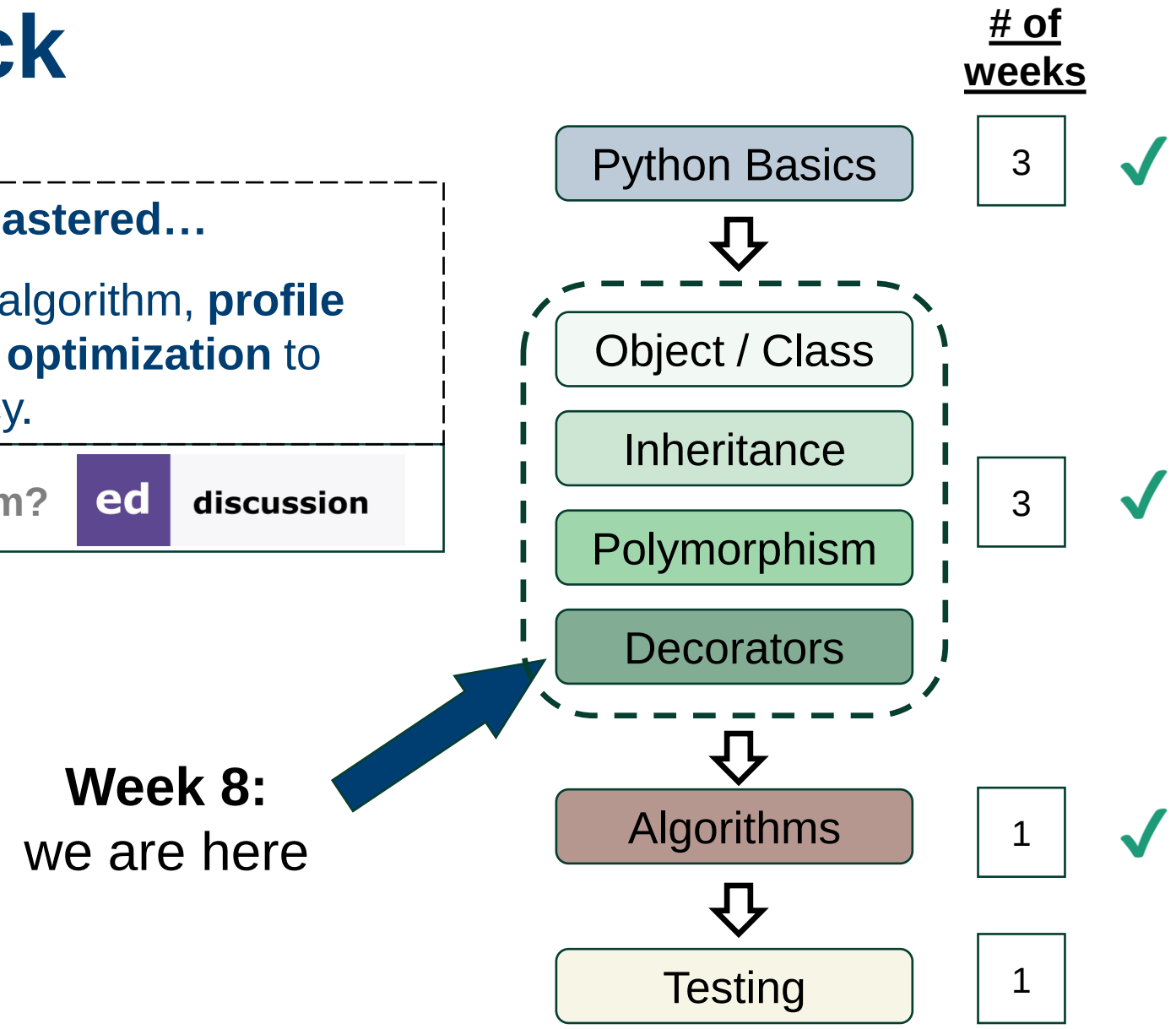
- String to string comparisons should work out, *in principle*.
  - Try out yourself!

Progress Check

Checklist: you should have mastered…

- How to **implement** a simple algorithm, **profile** (time) the code, and perform **optimization** to improve the runtime efficiency.

Questions outside the classroom? ed discussion



Decorators

A **decorator** is a special type of function that is used to modify the behaviour of another function or method.

Example from debug_timer.py

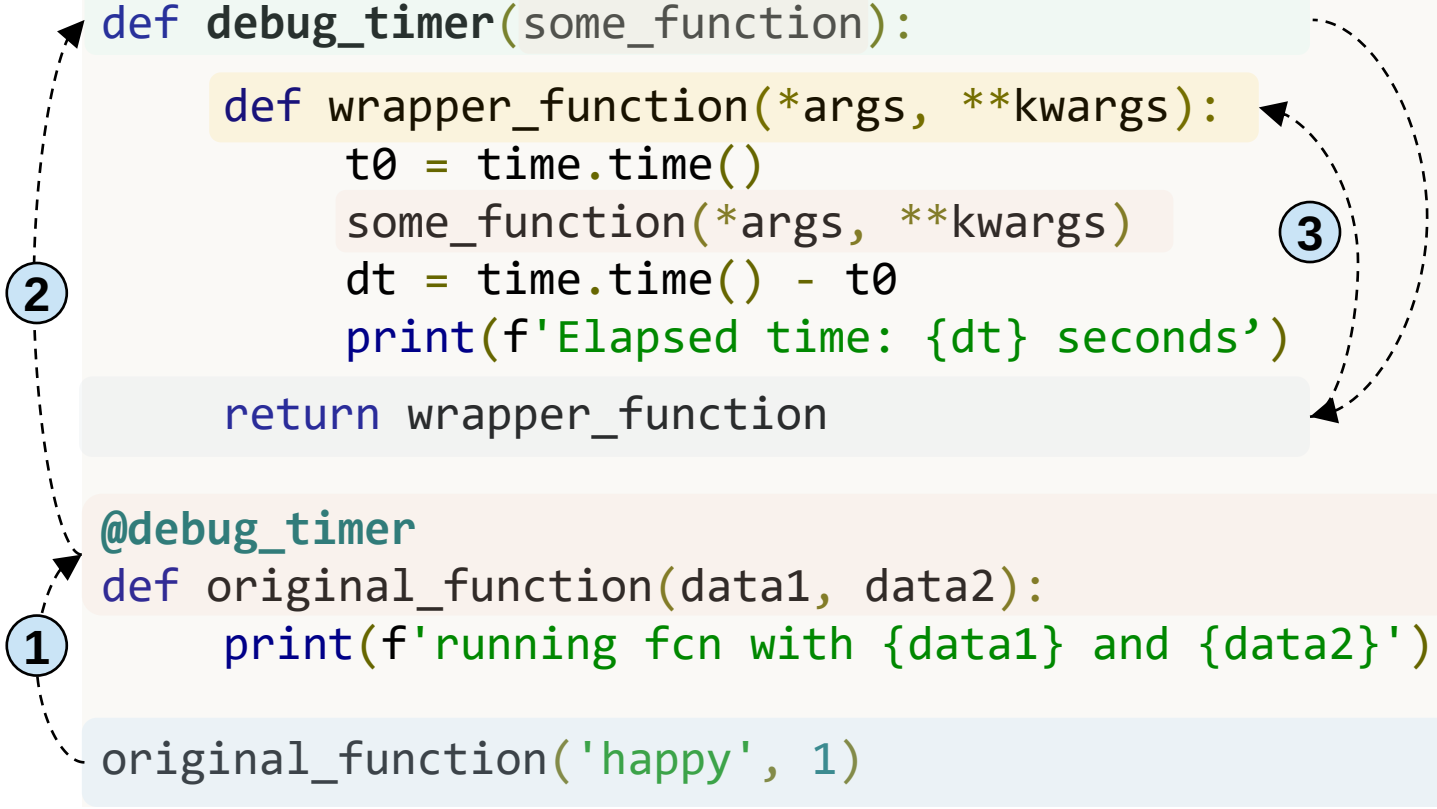
```
def debug_timer(some_function):

    def wrapper_function(*args, **kwargs):
        t0 = time.time()
        some_function(*args, **kwargs)
        dt = time.time() - t0
        print(f'Elapsed time: {dt} seconds')

    return wrapper_function

@debug_timer
def original_function(data1, data2):
    print(f'running fcn with {data1} and {data2}')

original_function('happy', 1)
```



1. **original_function** is called with the arguments **'happy', 1**
2. **original_function** is decorated with **@debug_timer**. When **debug_timer** invoked from **original_function**, **some_function = original_function**
3. **debug_timer** calls **wrapper_function** by revoking the `return` statement: **some_function** will be executed, as well as being timed

Static Methods

- In OOP, we can define a function that **does not rely on any instance attributes**
  - Utility functions
  - Basis functions (or, immutable things)

- There are **two possible ways** to implement such a function:
  - Using a normal function (defined outside of the class)
  - Using a static method (**`@staticmethod`**, defined within the class)

- Example: check if someone's age > 18.

Example 1: using a standalone function

```
class Person:
    def __init__(self, age):
        self.age = age;
        self.adult = is_adult(age);

def is_adult(age):
    return age > 18;
```

Example 2: using a static method

```
class Person:
    def __init__(self, age):
        self.age = age;
        self.adult = self.is_adult(age);

    @staticmethod <----- decorator (must have)
    def is_adult(age):<----- no need to inc. self
        return age > 18;
```

revoke with `self`

Class Methods

- In OOP, we are allowed to instantiate a new object **in two ways**
  - Directly calling the class constructor
  - Using a class method (**`@classmethod`**, a method defined within the class)

- Example: calculating age from birth year

- A broader usage: class methods can modify class attributes

Example

```
from datetime import date

class Person:
    def __init__(self, age = 0):
        self.age = age

    @classmethod  <----- decorator (must have)
    def fromBirthYear(cls, year):
        return cls(date.today().year - year)
```

1. Return the calculated age
2. Construct a new class
3. Assign to `self.age`

Driver code

```
p1 = Person(20)
print(p1.age)

p2 = Person.fromBirthYear(2004)
print(p2.age)
```

The same effects!

Your task today

Four mini-tasks, featuring the exercises of

- **Animation with cmd/Terminal**
- **Decorators and Wrapper functions**
- **Static method, class method, and property function**

To start…

- Study the Python scripts from your Friday lecture.
- Read and study the sample output from the lab sheet carefully.
- Revise the **Command Prompt / Terminal commands** listed in the Lab 2 sheet and slides.

Questions?

That's it for now.

You can now proceed to the Lab 8 exercises.