IMPERIAL

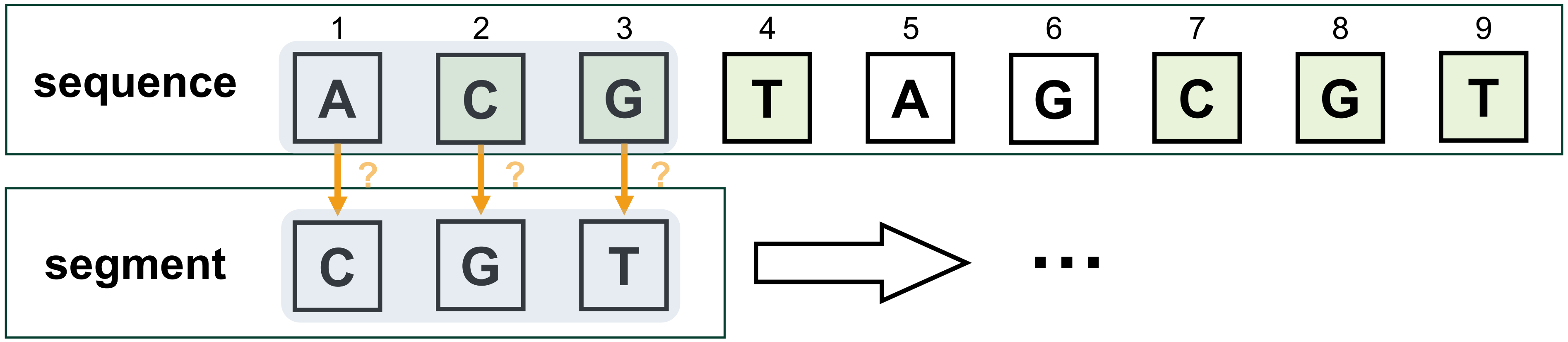
BIOE50010 – Programming 2

Computer Lab 8: Advanced Function & Class Mechanics

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

23 November, 2025

Feeback on Week 7 - `find()`



- There are many possible ways to structure the `find()` algorithm:
  - **Character-to-character** comparison: uses 2 nested `for`-loops, *slow*.
  - **List-to-list** comparison: uses 1 `for`-loop to slice the sequence, *faster*.
  - **List-to-list** comparison **with an initial-character check**: Avoids unnecessary slicing by checking the first character first, *fastest*.
- String-to-string comparisons should work – strings are iterable.

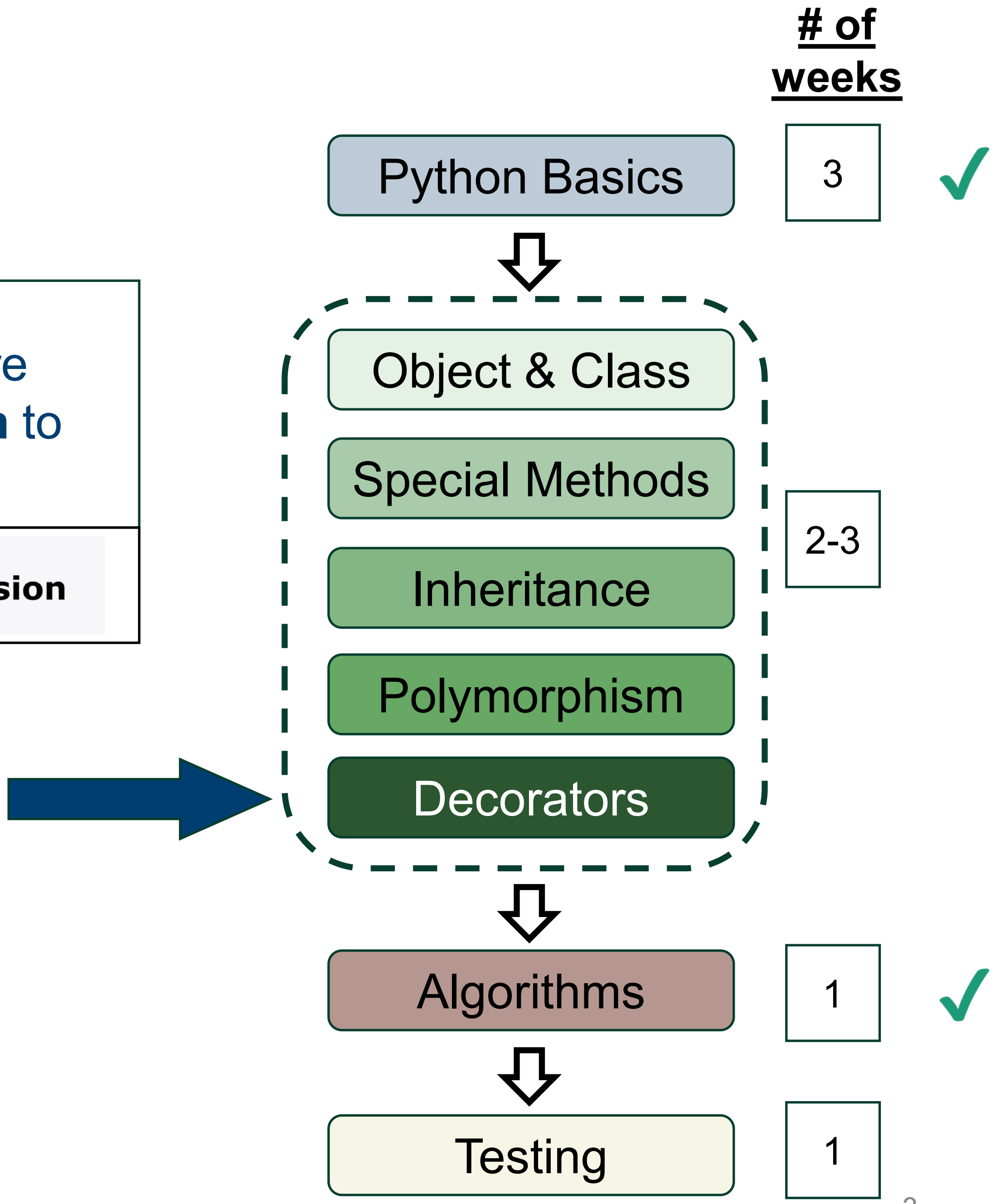
Progress Check

Revision Points (from weeks 7)

- How to **implement** a simple algorithm, measure your code efficiency, and perform **optimization** to improve its efficiency.

Questions outside the classroom? **ed discussion**

Week 8:
we are here



Function Decorators

- A **decorator** is a special type of function that is used to modify the behaviour of another function.
  - Wrapper functions
  - Static method (in OOP)
  - Class method (in OOP)
  - Property method and setter method (in OOP)
  - …
- When a function (or, method) is ***decorated***, we place an `@` symbol directly above a function.

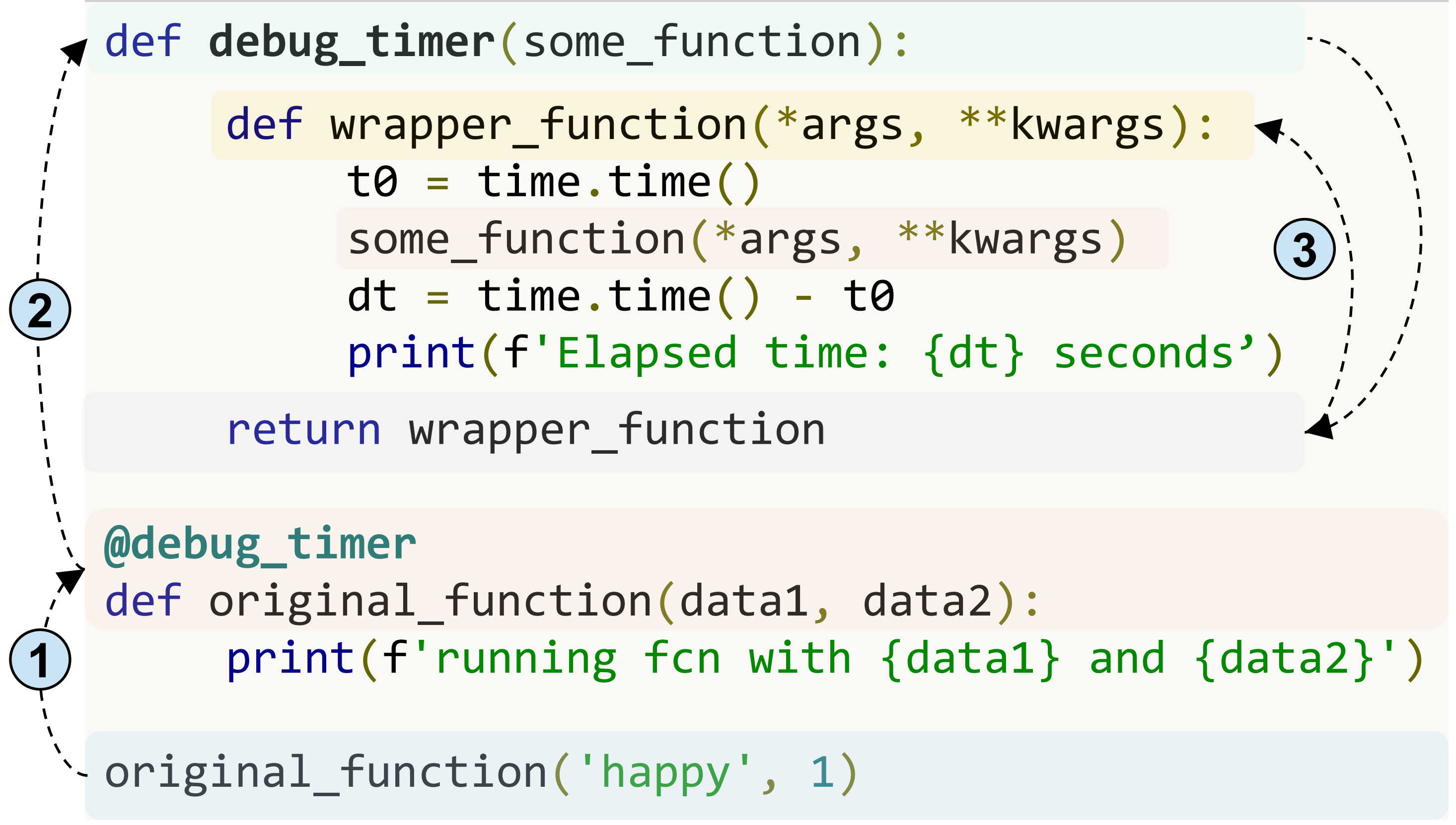
```
@myDecorator
def myFunction():
     ...
```

- It means: `myFunction = myDecorator(myFunction)`
  - In this case, a function (`myFunction`) is passed into another function (`myDecorator`) as an argument.

Wrapper Functions

- Rule 1: a function can be passed into another function as an argument.
- Rule 2: a function can be defined in another function.

Example from debug_timer.py



```python
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: so now, the argument, **some_function**, will be executed, as well as being timed.

* See weekly coding example here.

`@staticmethod`

- Sometimes we want a method (in OOP) that **does not use any instance data.**
  - *i.e.*, no need access to `self`.
- Such methods are useful for:
  - Utility functions.
  - Operations that don't use object state.
- There are two ways to define them:
  - A regular function defined **outside** the class.
  - A `@staticmethod` defined **inside** the class.

Example: check if someone's age > 18. Using a regular function or a static method works the same way functionally.

* See weekly coding example here.

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

@classmethod

- In OOP, we are allowed to instantiate a new object **in two ways**:

1. Directly calling the class constructor.
2. Using a class method (`@classmethod`) method as an alternative constructor.

Example: calculating someone's age from his/her birth year:

- ❑ Call the class method using the birth year
- ❑ The method calculates the age
- ❑ The calculated age is passed to the constructor
- ❑ The constructor assigns the value to `self.age`

Example

```python
from datetime import date

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

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

Driver code

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

p2 = Person.fromBirthYear(2005)
print(p2.age)
```

$\}$ The same effects!

* See weekly coding example here.

Your Tasks Today

Four short tasks combining use of procedural programming and object-oriented programming:

- **Computer animation** in Command Prompt (Windows PCs) / Terminal (Mac).
- Use **wrapper functions** to time your code.
- **Decorators in classes**: static method, class method, and property function.

To start…

- Study the syntax using the Python snippets from your Friday lecture slides and weekly example notebook.
- Read the sample output from the lab sheet carefully.
- Revise the **Command Prompt / Terminal commands** listed in the Lab 2 sheet and slides.