IMPERIAL

BIOE50010 – Programming 2

Computer Lab 4

Binghuan Li Department of Chemical Engineering

Maria Portela Department of Bioengineering

Wenhao Ding Department of Bioengineering

25 October, 2024

Alignment in Formatting

5 spaces

```
-2    . . .
-1    . . .
0     . . .
1     . . .
2     . . .
```

- Suppose the desired effect from formatting is
- … which can be achieved by

Example

```
for i in range(-2, 3):
    print(f"{i:<5}", end='')
    for j in range(5):
        print(" . ", end='')
    print()
```

The key trick here is `"{i:<5}"`

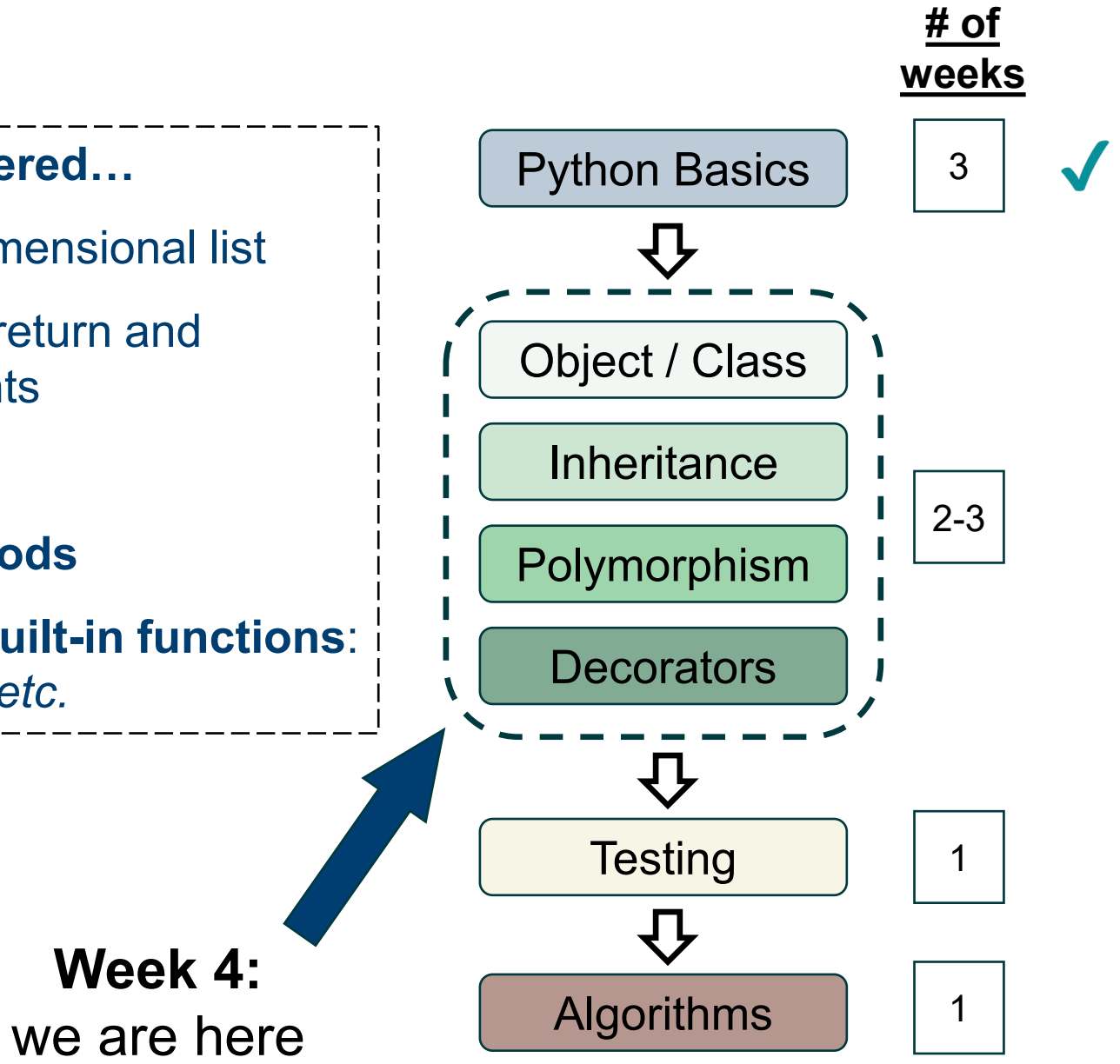
- `:<` tells Python to **left-align** the text
- **5** is the **width** of the space allocated for the text (5 characters wide inc. the contents to be printed).

- Similarly, we can right-align the texts (`:>`) , centrally align the texts (`:^`), or auto-fill the empty spaces (`:`*`char`*`>`), more examples on [Python f-string cheat sheet]

Progress Check

Checklist: you should have mastered…

- **Data structures**: str, int, high-dimensional list
- **Functions,** namespaces, using return and keyword / non-keyword arguments
- **File I/O**: read and write
- Python built-in **string / list methods**
- Other commonly-used **Python built-in functions**: `range()`, `enumerate()`, `len()`, *etc.*



Object-Oriented Programming

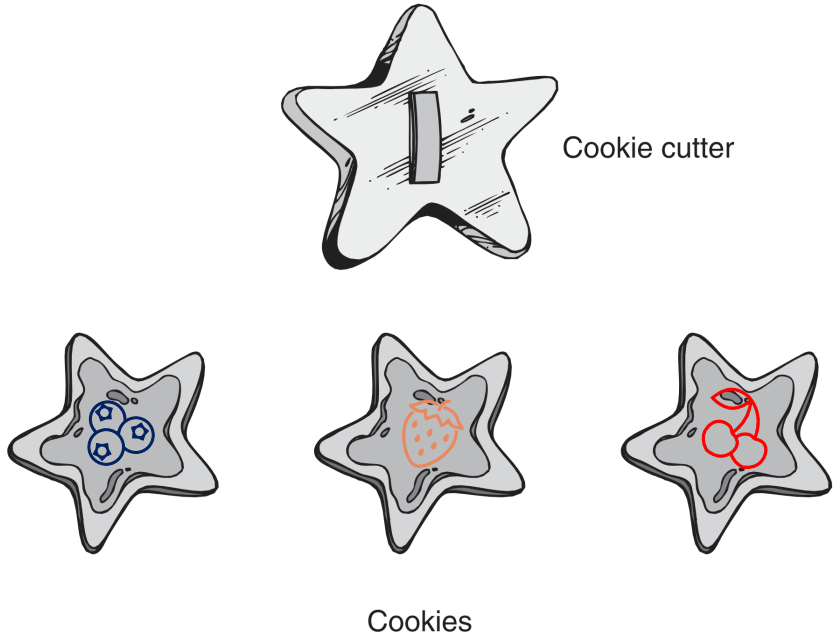
new terminology!

- Two most commonly-used programming paradigms:
  - **Procedural** (aka what you have done so far): programs are composed of one or more functions, executed serially;
  - **Object-oriented**: programs based on the **objects**, where data and functions are 'packed' into a **user-defined data structure**.
- **Examples of objects**: a `str`, `list`, `dict`…
  - These are the **data structures**, rather than the real data!
- The prototype / blueprint of an object is structured by the **class definition**.

A ***cookie cutter*** can be used to make different ***cookies***.

⟺

A ***class definition*** can be used to make several ***objects***.



Source: Starting Out with Python, 4th Ed.

- Sometimes, **objects** are also referred to as the **instances**.

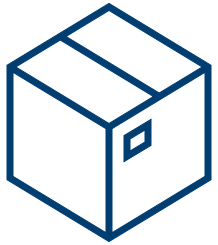
What Does an Object Hold?

new terminology!

an object

variables — *a.k.a.* **attributes**

functions — *a.k.a.* **methods**



An object ‘packs’ (**encapsulates**) variables and functions

Suppose we have a box…

- The colour is red
- The state is closed

} attributes

… and I can do the following things to manipulate the property / state of the box:

- Open the box
- Describe its properties
- Close the box
- Fold it…

} methods

Example Code (1/)

`self`: an identifier refers to the object *itself*, provides access to attr. / methods

new terminology!

Example

```
class Box:
    def __init__(self, color):
        self.color = color
        self.is_open = False

    def describe_box(self):
        print(f"This is a {self.color} box.")

    def open_box(self):
        if not self.is_open:
            self.is_open = True
            print(f"The {self.color} box is now open.")
        else:
            print(f"The {self.color} box is already open.")
```

attributes

methods

Describe the properties / states *etc.* of the object

Manipulate the behaviours the of the object

See `Box_example.py` on Bb

Example Code (2/)

constructor: `__init__()` is triggered automatically when the object is **instantiated**.

new terminology!

Example

```
class Box:
    def __init__(self, color):
        self.color = color
        self.is_open = False

    def describe_box(self):
        print(f"This is a {self.color} box.")

    def open_box(self):
        if not self.is_open:
            self.is_open = True
            print(f"The {self.color} box is now open.")
        else:
            print(f"The {self.color} box is already open.")
```

Driver

```
box = Box(color="blue")
box.describe_box()
box.open_box()
box.open_box()
```

Console

```
This is a blue box.
The blue box is now open.
The blue box is already open.
```

See `Box_example.py` on Bb

Your task today

Create a class **`Point`** that handles operations on Cartesian coordinates (x, y)

- Display the coordinates
- Convert (x, y) to polar coordinates (r, θ)
- Implement operator overloading *e.g.* addition, subtraction, multiplication…

To start…

- Take advantage of the code skeleton from the Friday live coding demonstration.
- Read all information and the sample output provided in the lab sheet carefully.
- Read sec. 17.5-17.8 in 'Think Python 2e' for **special methods** and **operator overloading**. *e.g.*, `__init__`, `__str__`, `__add__`, `__radd__`

Operator Overloading

- A same operator can have different behaviors when it is applied to different data types. For example, with the '**+**' operator;

`int + int`: arithmetic addition

`3` ← `1 + 2`

`str + str`: concatenation

`'ab'` ← `'a' + 'b'`

- **Operator overloading** enables users to define the rules of an operator when it is applied to the user-defined data types. *e.g.*, `+`, `-`, `*`, `==`, `<=`

`Point + Point`: what will happen?

`(4, 6)` ← ? `(1, 2) + (3, 4)`

- In this situation, the rule(s) for '`+`' need to de defined with the special (magic) method `__add__` in `Point`

Questions?

That's it for now.

You can now proceed to the Lab 4 exercises.