Data Driven World
Week 4

Simon Perrault– Singapore University of Technology and Design

Week 4 – Object Oriented Programming

a.k.a “when we lost half of the class”

Object?

- How to represent complex data?
- Example: Student
- What “defines” a student?

What defines a “Student”?

- **Grades**
- Name
- Student ID
- Pillar

How to represent a unique student?

Representing a student

- Solution 1: using a list

```
['Simon', '100XXXX', 'ISTD', [78, 62, 47, 98]]
```

name student id pillar grades

How to compute a student's average grade?

Manipulating Complex Data

- To compute average grade, we can do a function:

```
def avg_grade (student):
    n = len(student[3])
    sum = 0
    for grade in student[3]:
        sum += grade
    sum /= n
    return sum
```

Problems

- Data Representation is rather arbitrary
  - Need to know which information is stored
  - And in which order
- Functions to perform operation on such data may be scattered around and unclear

Instead…

- Let us define a new “class” of data
- A class is a **blueprint**, it stores “specifications”

```
class Student:
      _name = ””
      _id = “”
      _pillar = ””
      _grades = []
```

Such variables are called “Attributes”

Relevant information is stored within a class.
Class acts like a blueprint and can contain variables
Variables tend to have a name starting with underscore

Constructor/initializer

- A function with a special name can be defined called `__init__`
- Rule of thumb: any special function starts and ends w/ 2 underscores
- The `__init__` function takes as many arguments as needed to create a valid instance
- There can be only one `__init__` per class (other languages allow for multiple)

Question

- How many input should __init__ take considering this class:

```
class Student:
      _name = ””
      _id = “”
      _pillar = ””
      _grades = []
```

Answer: 5!

__init__ for Student:

```
class Student:
        _name = ""
        _id = ""
        _pillar = ""
        _grades = []

        def __init__ (self, name, id, pillar, grades):
                self._name = name
                self._id = id
                self._pillar = pillar
                self._grades = grades

simon = Student('Simon', '100XXXX', 'CSD', [78, 62, 47, 98])
```

self is an implicit argument/input. **You do not need and should not provide it**

__init__ should not be called explicitely. Instead use the name of the class

What if…

- …a student is created without any grades?

```
def __init__ (self, name, id, pillar, grades=[]):
        self._name = name
        self._id = id
        self._pillar = pillar
        self._grades = grades

paul = Student(”Paul”, “100YYYY”, “ASD”)
```

You can provide a default value for any argument

Other Functions within a Class

- A function within a class is called a “method”

```
class Student:
        name = ””
        id = “”
        pillar = ””
        grades = []
        [missing __init__ for lack of space]
        def avg_grade (self):
                n = len(self.grades)
                sum = 0
                for grade in self.grades:
                        sum += grade
                return sum / n
```

self refers to the student object calling the method. Note: the method does not take any additional input

self.grades refers to the attribute/variable called grades within the student object calling the method

Calling a method

```
Class Student:
    [refer to the code from the last few slides]


simon = Student('Simon', '100XXXX', 'CSD', [78, 62, 47, 98])
print(simon.avg_grade())
# self will be equal to simon when invoking
```

Summary

- Attributes are variables stored within a class
- Constructor/initializer is a specific function called __init__ which can create new instances of the class it belongs to
- Methods are functions within classes
  - Their first argument is always self
  - Self is implicit and should not be provided

Advantages

- Any relevant information (variable, function) **linked to a concept is contained** within a class
- The object-oriented philosophy is that every object is a **blackbox** with a defined set of methods
  - **we do not know or need to know** how the data is stored inside the class
  - It’s fine to use attributes within the class, but avoid doing so outside

Property

```
# Class definition
class RobotTurtle:
    # Attributes:
    def __init__(self, name, speed=1):
        self.name = name
        self.speed = speed
        self._pos = (0, 0)

    # property getter
    @property
    def name(self):
        return self._name

    # property setter
    @name.setter
    def name(self, value):
        if isinstance(value, str) and value != "":
            self._name = value
```

- Robot is created with a name and speed
- Property is a family of methods:
  - getter (retrieve value)
  - setter (set value)

Getter or Setter?

Getter

```
@property
def name(self):
    return self._name
```

1. One argument: `self`
2. Used to return the value of an attribute

Setter

```
@name.setter
def name(self, value):
    self._name = value
```

name here is the name of the attribute without a _

1. Two arguments: `self` & `value`
2. Used to set the value of an attribute

Both require the @property decorator before the function definition!

Advantages

Getter

- You can control the return value of the function, and e.g. decide not to return the full value

```
@property
def nric(self):
    return self._nric[-4:]
```

Setter

- You can check the value provided as input and control it

```
# property setter
@name.setter
def name(self, value):
    if isinstance(value, str) and value != "":
        self._name = value
```

If one attribute has a getter (property) but no setter, the attribute is "de facto" read only

Access to Property/Computed Property

- For getters (or properties), you could also return a computed value

```
class Coordinate:

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    @property
    def distance(self):
        return math.sqrt(self.x * self.x + self.y * self.y)
```

```
# object instantiation
p1 = Coordinate(3, 4)
print(p1.x, p1.y)
print(p1.distance)
```

A property (or getter) is accessed like an attribute

Access to setter

- Similarly to property, access to a setter is similar to an attribute
- Example of name setter from the Robot class:

```
# property setter
@name.setter
def name(self, value):
    if isinstance(value, str) and value != "":
        self._name = value
```

```
my_robot.name = "T4new"
print(my_robot.name)
my_robot.name = ""
print(my_robot.name)
```

```
T4new
T4new
```

The second setter fails because the string is empty (see setter code)

Note on Attributes

- Note: you do not have to “declare” the attributes in the class
- They can be dynamically “created” in any method

```
class Student:
    def __init__ (self, name, id, pillar, grades):
        self._name = name
        self._id = id
        self._pillar = pillar
        self._grades = grades
```

All four attributes will be created after instantiation

Checking if a variable is of a given type

- Sometimes, we may want to check whether a variable is indeed an instance of a class

```
simon = Student('Simon', '100XXXX', 'CSD', [78, 62, 47, 98])
isinstance(simon, Student)
>>> True
isinstance(simon, Robot)
>>> False
```

isinstance: takes 2 arguments (variable name and type)
Returns whether the variable is of the specified type

Composition

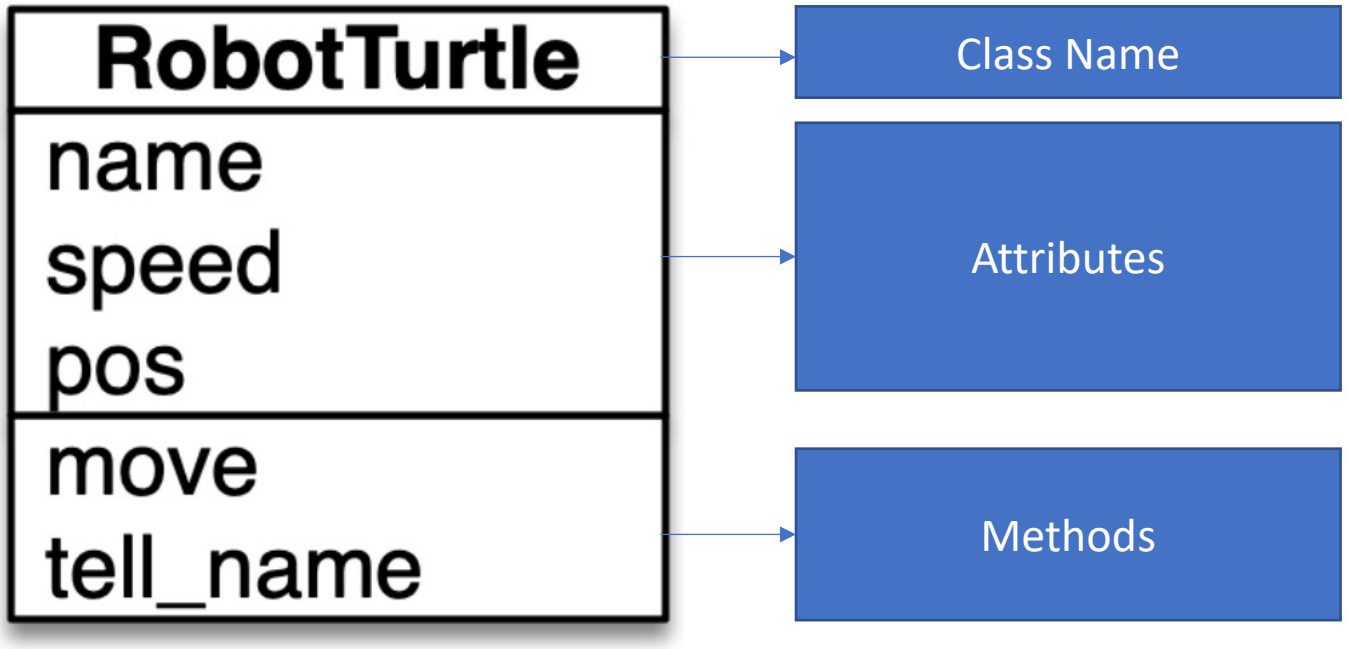
- See Week 4 Notes ☺
- https://github.com/Data-Driven-World/d2w_notes/blob/master/Object_Oriented_Programming.ipynb

Special Methods

- Similarly to `__init__` there are several useful special methods
- `__str__`
  - Takes one argument: `self`
  - A representation of an instance as a string.
  - Called when the object is printed (e.g. `print(simon)`)
- `__cmp__`
- Takes two argument: `self` and `others`
  - Used to compare the current instance `self` with another instance `others`
  - Returns a number (usually -1, 0 and 1)
  - Used to compare with <, >, ==, !=, >=, <=

UML Representation

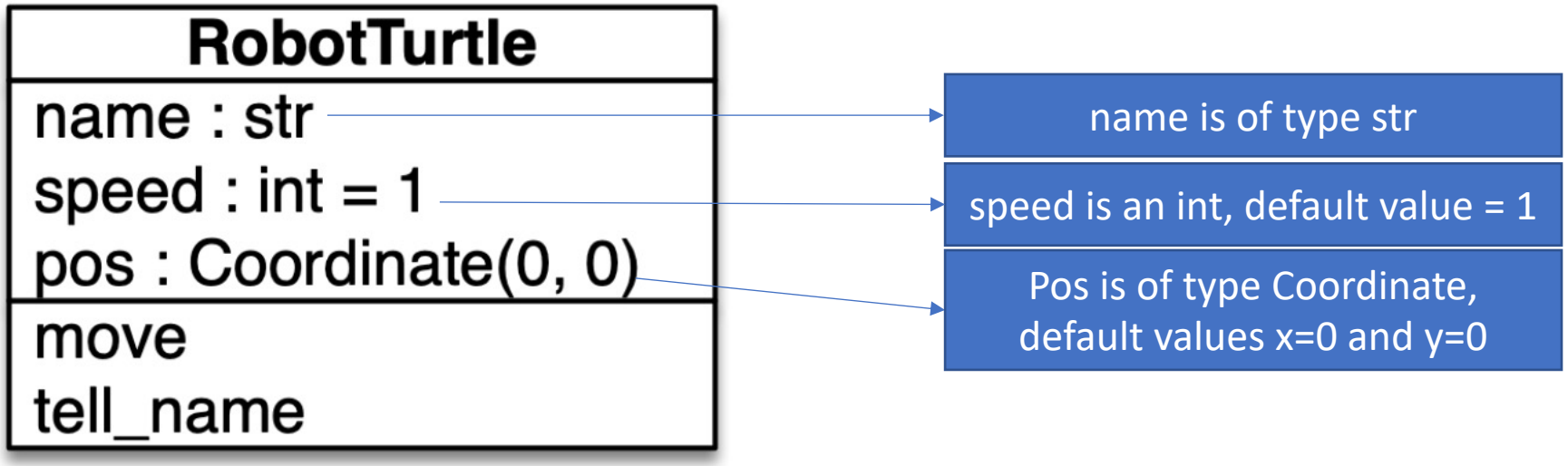
- To represent a data type (class) we use UML Diagrams.
- **Simple version:**



Note: in most cases, special methods are not shown

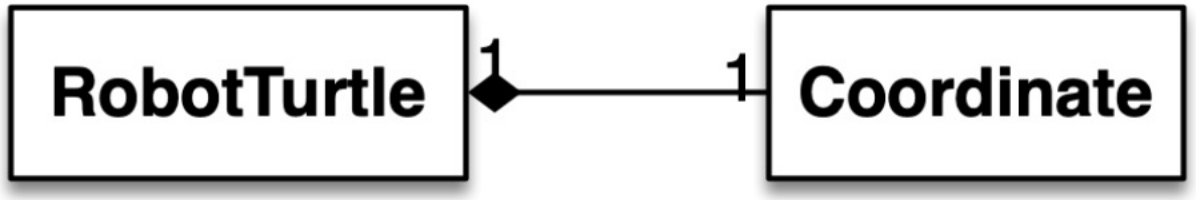
UML Representation (Advanced)

- More advanced:



Relationship between Classes

- UML can be used to show relationship between classes



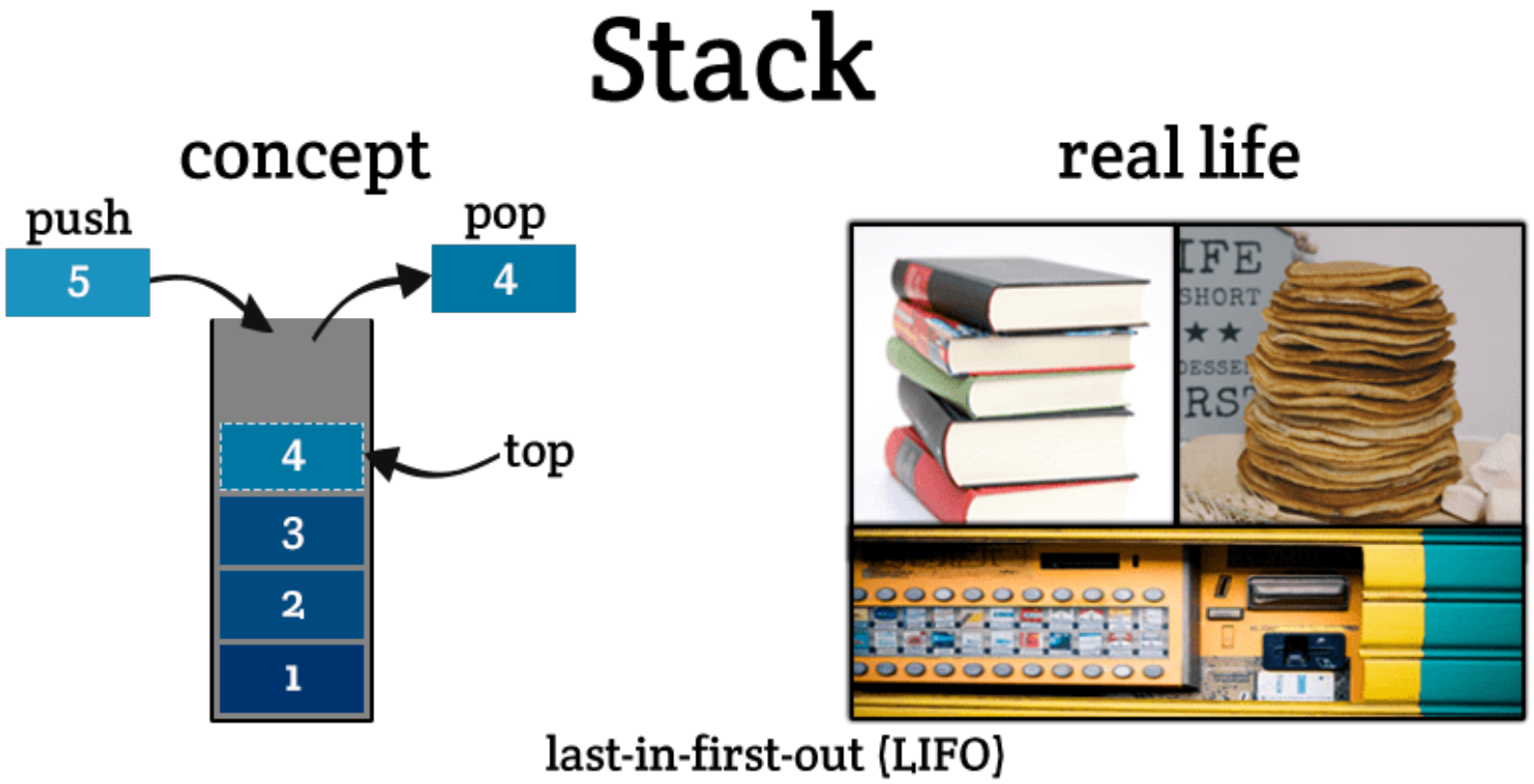
- The black diamond shows a composition relationship, i.e. an instance of RobotTurtle contains an instance of Coordinate
- The “1” on both sides show cardinality:
  - 1 instance of Robot is associated with exactly 1 instance of Coordinate

Week 4 – Stack and Queues

“Q: What do you call a group of cats in a queue? - A: A Feline”

Stack

- Last-in-first-out (LIFO): the last object (top) Is the first to be removed

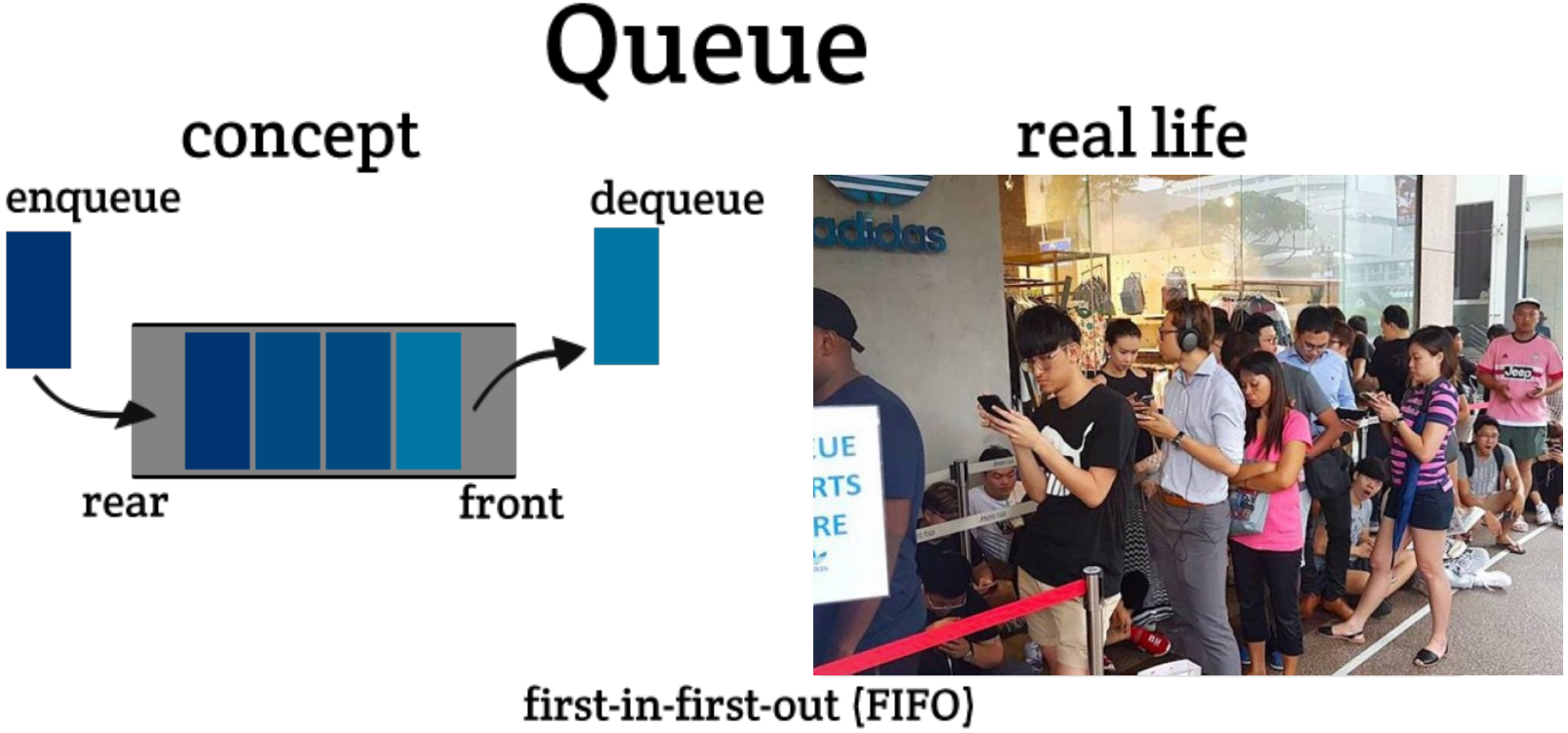


Stack Attribute and Methods

- Attributes:
  - List containing all the items stored in the Stack
- Method:
  - `push`: method to put an item on top of the stack (at the end of the list)
  - `pop`: method to retrieve the **last item**, and remove it from the stack
  - `peek`: method to retrieve the **last item**, without removing it from the stack

Queue

- First-in-first-out (FIFO): the first item to arrive is the first to be leave



Queue Attribute and Methods

- Attributes:
  - List containing all the items stored in the Stack
- Method:
  - `push`: method to put an item in the queue (at the end of the list)
  - `pop`: method to retrieve the **first item**, and remove it from the queue
  - `peek`: method to retrieve the **first item**, without removing it from the stack

Similarities

- Both use a list to store data
- Both have the same set of methods with similar behavior
- **Think of a name that can describe both concepts**