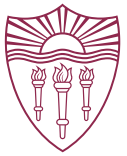
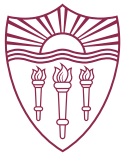


DSCI 510
PRINCIPLES OF PROGRAMMING FOR DATA SCIENCE

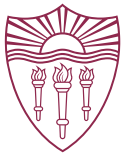
Itay Hen



OOP

OBJECT ORIENTED PROGRAMMING (CONTINUED)

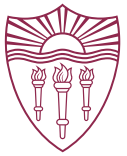
dir() and type()



```
>>> l = list()
>>> type(l)
<class 'list'>
>>> dir(l)
['__add__', '__class__', '__class_getitem__',
'__contains__', '__delattr__', '__delitem__', '__dir__',
'__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__getitem__', '__gt__', '__hash__',
'__iadd__', '__imul__', '__init__', '__init_subclass__',
'__iter__', '__le__', '__len__', '__lt__', '__mul__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__reversed__', '__rmul__', '__setattr__',
'__setitem__', '__sizeof__', '__str__',
'__subclasshook__', 'append', 'clear', 'copy', 'count',
'extend', 'index', 'insert', 'pop', 'remove', 'reverse',
'sort']
```

- The dir() command lists capabilities.
- The methods starting with '_' are conventionally known as magic/dunder methods. You can implement magic methods for your classes to make them behave as Python objects.
- In contrast, methods that do not start with '_' are considered “regular” or “user defined” methods.
- The type() method prints the type of the object.

dir() and type()

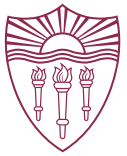


```
class PartyAnimal:
    x = 0

    def party(self):
        self.x += 1
        print(f'So far: {self.x}')


pa = PartyAnimal()

print(type(pa))
<class '__main__.PartyAnimal'>
print(dir(pa))
```

['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', **'party'**, **'x'**]

- You can use dir() to find "capabilities" of our newly created class.
- You can implement the magic methods to make your class behave like Python objects.

__dict__



- __dict__ is a **built-in attribute** that stores an object's **namespace** — i.e., all of its writable attributes and their current values.
- It's a dictionary that maps attribute names (as strings) to their values for that object.

python

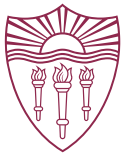
```python
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age

fido = Dog("Fido", 5)

print(fido.__dict__)
```

Output:

python

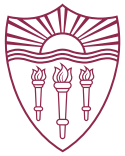
```
{'name': 'Fido', 'age': 5}
```

Object Lifecycle



- Objects are created, used, and discarded
- We have special blocks of code (”magic” or “dunder” methods) that get called
  - At the moment of creation (constructor)
  - At the moment of destruction (destructor)
- Constructors are used a lot
- Destructors are seldom used

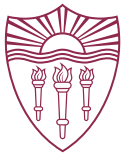
Constructor



- In object-oriented programming, a *constructor* in a class is a special block of statements called when an object is created.

- The primary purpose of the *constructor* is to set up some instance variables to have the proper initial values when the object is created.

Object Constructor and Destructor



```
class PartyAnimal:
    x = 0

    def __init__(self):          Constructor
        print('I am constructed')

    def party(self):
        self.x += 1
        print(f'So far: {self.x}')

    def __del__(self):           Destructor
        print(f'I am destructed: {self.x}')
```

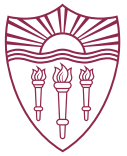
```
pa = PartyAnimal()
I am constructed

pa.party()
So far: 1

pa.party()
So far: 2

pa = 42
I am destructed: 2

print(f'pa: {pa}')
pa: 42
```

Constructor is typically used to initialize instance attributes.

Destructor is called when an object is deleted by the garbage collector.

The timing of when the *destructor* gets called is **not guaranteed.**

Let's talk about self



Always include *self* as the first parameter in instance methods of a Python class!

self is a keyword in Python **(True/False)** **False**

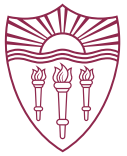
```
class PartyAnimal:
    x = 0
    name = ""

    def __init__(myself, name):
        myself.name = name
        print(f'{myself.name} constructed')

    def party(myself):
        myself.x += 1
        print(f'{myself.name} party count: {myself.x}')
```

self is used as a convention in Python.

You can use any string in its place.

But you should adhere to the convention as it improves code consistency and makes it easier for others to understand your code.

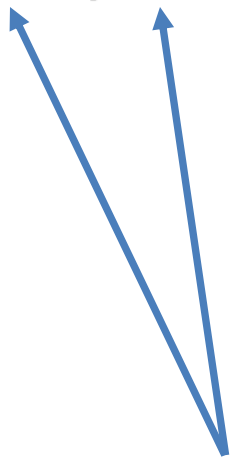
Many Instances



- We can create lots of *objects* - the class is the template for the object
- We can store each distinct *object* in its own variable
- We call this having multiple *instances* of the same class
- Each *instance* has its own copy of the **instance attributes**.
- Each *instance* shares the **class attributes**.

A 2-D Vector Class

```
class Vector:

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

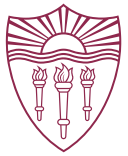


Default values in case we initialize without passing values.

What operations are supported by Vectors?

1. You can add two vectors.
2. You can compute the absolute value of vectors
3. You can multiply a vector with a scalar value.

A 2-D Vector Class



```
v1 = Vector(3, 4)
v2 = Vector(5, 7)
```

```
v3 = v1 + v2
```

TypeError: unsupported operand type(s) for +: 'Vector' and 'Vector'

```
v4 = v1 * 4
```

TypeError: unsupported operand type(s) for *: 'Vector' and 'int'

- Our Vector class is not very useful at this point.
- We will implement some magic methods to emulate numeric types.
- Namely, implement,
  - __add__ to support vector addition
  - __mul__ to support multiplication
  - __abs__ to calculate the magnitude of the vector

A 2-D Vector Class

We'll discuss *import* a little later

```
import math


class Vector:

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

    def __abs__(self):
        return math.hypot(self.x, self.y)

    def __add__(self, other):
        x = self.x + other.x
        y = self.y + other.y
        return Vector(x, y)

    def __mul__(self, scalar):
        return Vector(self.x * scalar, self.y * scalar)
```

To get the magnitude of a vector

To support vector addition, returns a new Vector.

To support multiplication of a vector, returns a new vector

```
v1 = Vector(3, 4)
v2 = Vector(5, 7)

v3 = v1 + v2
print(v3)
```

<__main__.Vector object at 0x105337d60> '

```
print(f'x: {v3.x}, y:{v3.y}')
```

x: 8, y:11

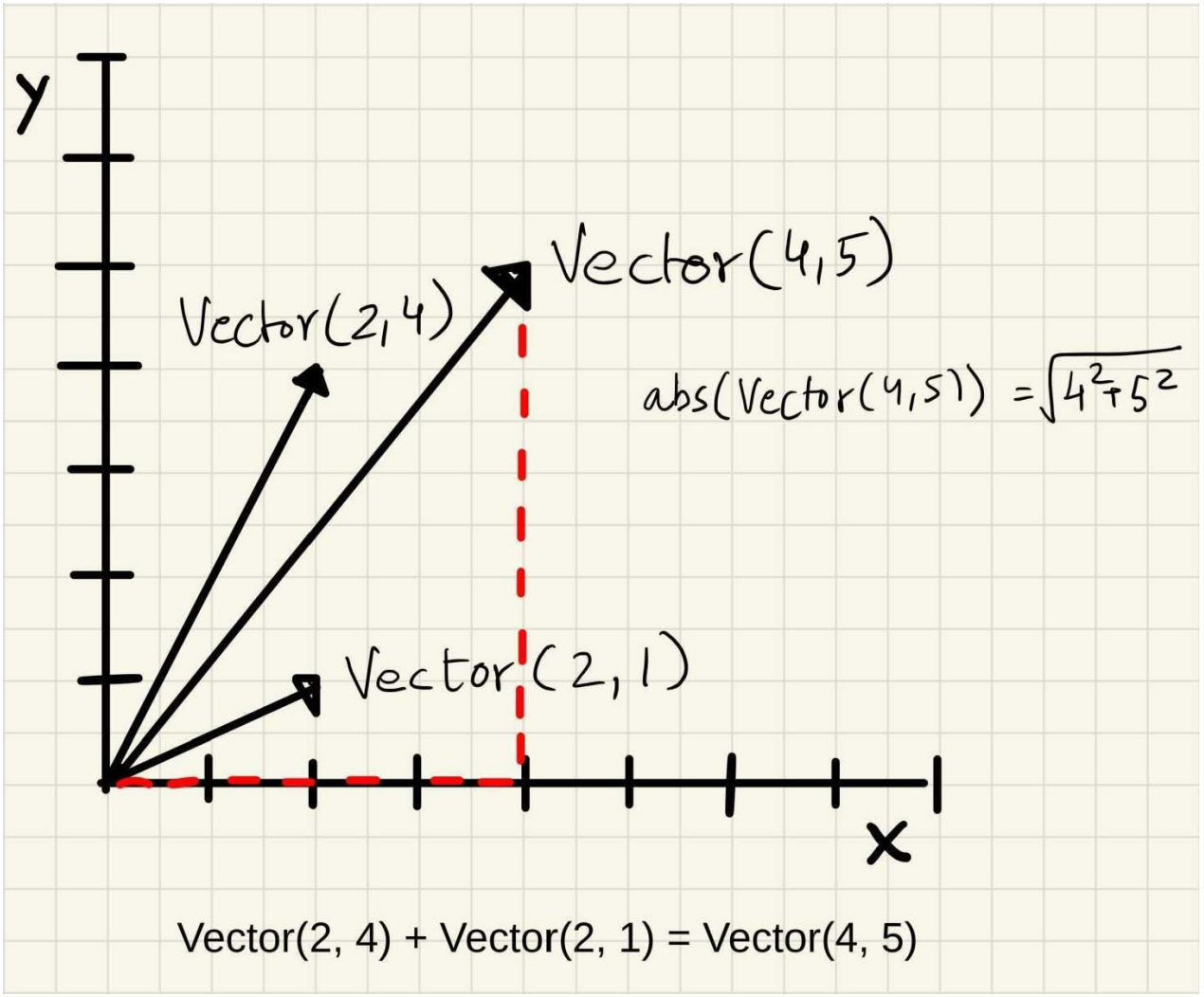
```
v4 = v1 * 4
print(f'x: {v4.x}, y:{v4.y}')
```

x: 12, y:16

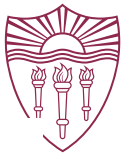
```
print(abs(v1))
```

5.0

2-D Vector Addition and Magnitude



String Representation of classes

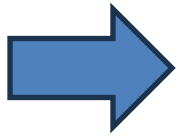


Both `repr()` and `str()` correspond to **dunder (double-underscore) methods** that you can define in your own classes:

Corresponding dunder method	Purpose
`obj.__repr__()`	Developer-facing, unambiguous *representation* of the object
`obj.__str__()`	User-facing, readable *string form* of the object

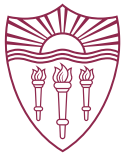
We can add the dunder __repr__ method to Vector:

```
def __repr__(self):
    return f"({self.x},{self.y})"
```

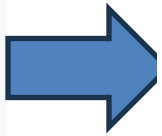


```
v1=Vector(2,3)
print(v1)
(2,3)
v2=Vector(4,5)
print(v2)
(4,5)
v3=v1+v2
print(v3)
(6,8)
```

String Representation of classes



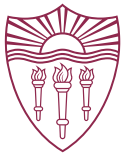
```
class Dog:
    def __repr__(self):
        return f"Dog(name={self.name!r}, age={self.age!r})"

    def __str__(self):
        return f"{self.name}, {self.age} years old"

    def __init__(self, name, age):
        self.name = name
        self.age = age

fido = Dog("Fido", 5)

print(repr(fido))  # Calls fido.__repr__()
print(str(fido))   # Calls fido.__str__()
print(fido)        # print() automatically calls __str__()
```



Output

```
Dog(name='Fido', age=5)
Fido, 5 years old
Fido, 5 years old
```

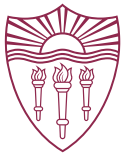
- When we print() an object, str() is called.
- If str() is not defined, then repr() is called.
- If neither, then default.



Of course, a language feature would not be worthy of the name “class” without supporting inheritance.

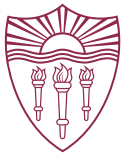
INHERITANCE

Terminology: Inheritance



- Definition:
  - Inheritance is a mechanism in which a new class (subclass) is created by inheriting properties (attributes and methods) of an existing class (superclass).
  - It promotes code reuse and establishes a relationship between the classes, allowing for a hierarchy of classes.
- Key Points:
  - **Parent (Superclass):** The original class whose properties are inherited.
  - **Child (Subclass):** The new class that inherits properties from the superclass and may extend or modify them.
  - 'Subclasses' are more specialized versions of the parent class

Inheritance in Python



The syntax for a derived class definition looks like this:

```
class DerivedClassName(BaseClassName):
    <statement-1>
    .
    .
    .
    <statement-N>
```

Method references are resolved as follows: the corresponding class attribute is searched, descending down the chain of base classes if necessary, and the method reference is valid if this yields a function object.

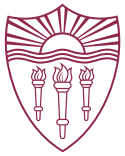
Derived classes may override methods of their base classes.

Python has two built-in functions that work with inheritance:

- Use isinstance() to check an instance's type: isinstance(obj, int) will be True only if obj.__class__ is int or some class derived from int.
- Use issubclass() to check class inheritance: issubclass(bool, int) is True since bool is a subclass of int. However, issubclass(float, int) is False since float is not a subclass of int.

```
>>> isinstance(v3,Vector)
True
```

Multiple Inheritance

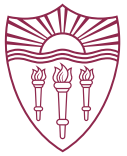


Python supports a form of multiple inheritance as well. A class definition with multiple base classes looks like this:

```
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    .
    .
    .
    <statement-N>
```

For the purpose of this class, that is all about multiple inheritance.

Inheriting from the PartyAnimal class



```
class PartyAnimal():
    x = 0
    name = ""

    def __init__(self, name):
        self.name = name
        print(f'{self.name} constructed')

    def party(self):
        self.x += 1
        print(f'{self.name} party count: {self.x}')


class FootballFan(PartyAnimal):
    points = 0

    def touchdown(self):
        self.points += 7
        self.party()
        print(f'{self.name} points: {self.points}')
```

- The class FootballFan derives from or extends the PartyAnimal class.
- It has all the capabilities of PartyAnimal and more

```
s = PartyAnimal('Sally')
```
Sally constructed

```
s.party()
```
Sally party count: 1

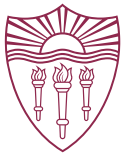
```
j = FootballFan('Jim')
```
Jim constructed

```
j.party()
```
Jim party count: 1

```
j.touchdown()
```
Jim party count: 2
Jim points: 7

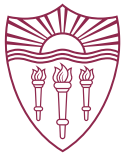
pythontutor.com

Another example



```
>>> class Vectorx0(Vector):
...     def __init__(self,y):
...         self.y=y
...         self.x=0
...
...
>>> issubclass(Vector,Vectorx0)
False
>>> issubclass(Vectorx0,Vector)
True
>>> v4=Vector(5)
Traceback (most recent call last):
  File "<pyshell#84>", line 1, in <module>
    v4=Vector(5)
TypeError: Vector.__init__() missing 1 required positional argument: 'y'
>>> v4=Vectorx0(5)
>>> print(v4)
(0,5)
>>>
```

Which class does PartyAnimal inherit from?



```
class PartyAnimal:
    x = 0
    name = ""

    def __init__(self, name):
        self.name = name
        print(f'{self.name} constructed')

    def party(self):
        self.x += 1
        print(f'{self.name} party count: {self.x}')
```

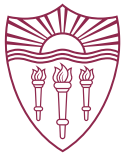
SAME AS

```
class PartyAnimal(object):
    x = 0
    name = ""

    def __init__(self, name):
        self.name = name
        print(f'{self.name} constructed')

    def party(self):
        self.x += 1
        print(f'{self.name} party count: {self.x}')
```

```
print(isinstance(PartyAnimal, object))
```

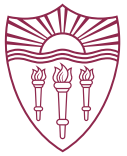
```
True
```

The object Class



- The object class is the root or base class for all classes in Python.
- Every class in Python implicitly inherits from object, even if it's not mentioned in the class definition.

Object Oriented Programming Definitions



Class: a template

Attribute: A variable within a class

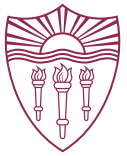
Method: A function within a class

Object: A particular instance of a class

Constructor: Code that runs when an object is created

Inheritance: Ability to extend a class to make a new class.

User Defined Objects



```
class Dog:
    def __init__(self, name):
        self.name = name

    def walk(self):
        return f'{self.name} *walking*'

    def speak(self):
        return f'{self.name} says Woof!'
```

```
1 usage
class IrishSetter(Dog):
    def __init__(self, name):
        Dog.__init__(self, name=name)

    def speak(self):
        return f'{self.name} says Arff!'

    def talk(self):
        return super().speak()
```

Not strictly necessary →

Example for overriding →

Calling a method of the superclass →

```
glenn = IrishSetter('Glenn')

print(glenn.walk())
```

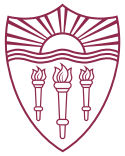
Glenn *walking*

```
print(glenn.speak())
```

Glenn says Arff!

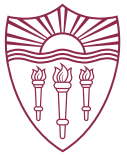
```
print(glenn.talk())
```

Glenn says Woof!



PYTHON MODULES AND LIBRARIES

What is a Module?



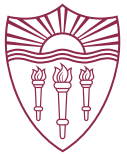
A module is a file containing Python code (functions, classes, variables) that can be imported and used in other Python programs.

Key Benefits:

- **Code Reusability:** Write once, use many times
- **Organization:** Keep related code together
- **Namespace Management:** Avoid naming conflicts
- **Maintainability:** Easier to update and debug

> **Simple Example:** A file named `math_utils.py` containing math functions is a module.

Types of Modules



1. Built-in Modules

Modules that come with Python installation:

- `math` - Mathematical functions
- `random` - Random number generation
- `datetime` - Date and time operations
- `os` - Operating system interface

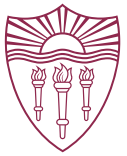
2. User-defined Modules

Modules created by you for your projects

3. Third-party Modules

Modules created by the community, installed via `pip`

Importing Modules



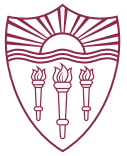
Different Ways to Import:

```
 # Import entire module
import math
print(math.sqrt(16))   # Output: 4.0

# Import specific items
from math import sqrt, pi
print(sqrt(16))   # Output: 4.0

# Import with alias
import numpy as np
array = np.array([1, 2, 3])

# Import all (not recommended)
from math import *
```

Not recommended because in this case Python imports **all** names (functions, constants, etc.) from the math module directly into our current namespace.

```
from math import *
pi = 3  # overwrites math.pi!
print(pi)  # 3 instead of 3.14159...
```

What is a Library/Package?



A **package** is a collection of related modules organized in a directory hierarchy.

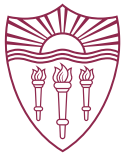
A **library** is a broader term referring to a collection of packages and modules.

Package Structure:

```
mypackage/
    __init__.py
    module1.py
    module2.py
    subpackage/
        __init__.py
        module3.py
```

> The `__init__.py` file makes a directory a Python package (can be empty).

Popular Python Libraries



Data Science & Analysis:

- **NumPy:** Numerical computing with arrays
- **Pandas:** Data manipulation and analysis
- **Matplotlib:** Data visualization

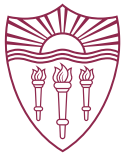
Web Development:

- **Django:** Full-featured web framework
- **Flask:** Lightweight web framework
- **Requests:** HTTP library

Machine Learning:

- **Scikit-learn:** Machine learning algorithms
- **TensorFlow:** Deep learning framework

Creating Your Own Module



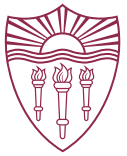
Step 1: Create a Python file (calculator.py)

```
def add(a, b):
    return a + b

def subtract(a, b):
    return a - b

def multiply(a, b):
    return a * b
```

Step 2: Import and use it

```
import calculator

result = calculator.add(5, 3)
print(result)  # Output: 8
```

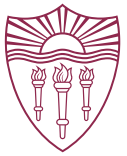
Installing Third-party Libraries



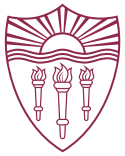
Using pip (Python Package Installer):

```
 # Install a library
pip install requests

# Install specific version
pip install requests==2.28.0

# Upgrade a library
pip install --upgrade requests

# List installed packages
pip list

# Uninstall a library
pip uninstall requests
```

Best Practices

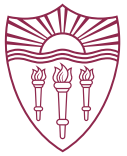


- **Import at the top:** Place all imports at the beginning of your file
- **Use meaningful names:** Module names should be lowercase and descriptive
- **Avoid circular imports:** Don't make modules depend on each other
- **Document your modules:** Include docstrings for functions and classes



RUNNING PYTHON FROM THE COMMAND LINE

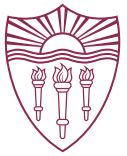
Running Python from the command line



- Python files can be executed directly from the command line.
- Command-line arguments allow dynamic input without modifying the code.
- Useful for scripts, automation, data processing, and testing.

- python myscript.py arg1 arg2 arg3
- python → invokes the interpreter
- myscript.py → file to execute
- arg1, arg2, etc. → arguments passed to the program

3. Accessing Arguments in Python



- First Import the sys module. This module provides access to system-specific parameters and functions that interact with the Python interpreter — for example, command-line arguments (sys.argv), the Python path (sys.path), and interpreter control (like sys.exit() or sys.version).
- Use sys.argv which is a list of command-line arguments.
- sys.argv[0] → script name
- sys.argv[1:] → user arguments

- Loop over sys.argv[1:] to print multiple arguments.
- Example: python script.py one two three → prints each on a new line.

Example

```python
# filename: greet.py
import sys

# Check if the user provided a name
if len(sys.argv) < 2:
    print("Usage: python greet.py <name>")
    sys.exit(1)

name = sys.argv[1]
print(f"Hello, {name}!")
```

How to run it:

```bash
python greet.py Itay
```

Output:

```
Hello, Itay!
```