Algorithms and Data Structures for Data Science

Object Oriented Programming

CS 277

Brad Solomon

January 25, 2024

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Department of Computer Science

Learning Objectives

Finish discussing function overloading

Introduce object-oriented programming

Discuss and practice defining interfaces for computational problems

An overview of common I/O formats in Python

Programming Toolbox: Function Overloading

Two functions are **overloaded** when they have the same name but different parameters.

```
def combine(x, y):
    return [x, y]

print(combine(5, 1))

def combine(list1, list2):
    return list1+list2

print(combine([1, 2], [3, 4]))

def combine(x, list1, list2):
    return [x]+list1+list2

print(combine(0, [1, 2], [4, 5]))
```

Programming Toolbox: Function Overloading

To properly define an overloaded function, give default arguments.

```
def combine(x, y=None, list1 = None, list2 = None):
    out = [x]
    if y:
        out+=[y]
    if list1:
        out+=list1
    if list2:
        out+=list2
    return out

print(combine(5, 1))


print(combine(0, [1, 2], [4, 5]))


print(combine(0, list1=[1, 2], list2=[4, 5]))


```

Programming Toolbox: Function Overloading

For true freedom of input, use keyword `*args` and `**kwargs`

```
def combine(*args, **kwargs):
    out = []

    for a in args:
        out.append(a)


    for k, v in kwargs.items():
        print("{} = {}".format(k, v))
        out+=v
    return out


print(combine(0, 1, 2, 3, 4, \
list1=[9, 2,3,1], list2=[8,7,2,1], \
list3 = [10]))
```

Programming Practice: Function Overloading

Write a function blackBox() that takes as input 0, 1, 2, or 3 arguments

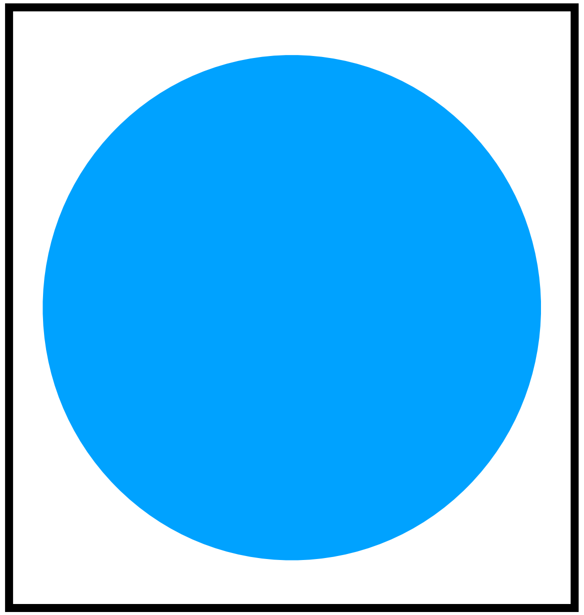
The default value for the first input arg is 3

The default value for the second input arg is 6

The default value for the third input arg is 0.5

Object-Oriented Programming

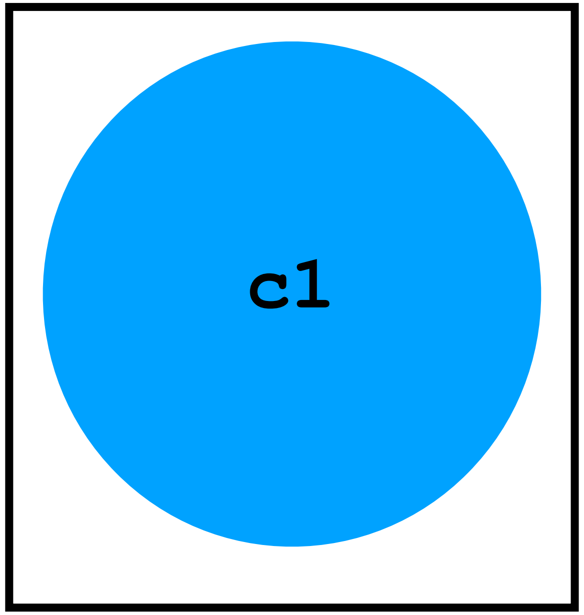
An **object** is a conceptual grouping of variables and functions that make use of those variables. A function associated with an object is a **method.**



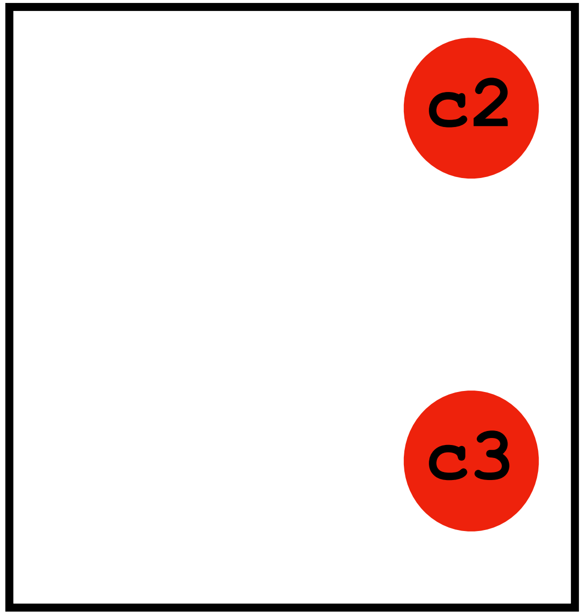
Variables:

Methods:

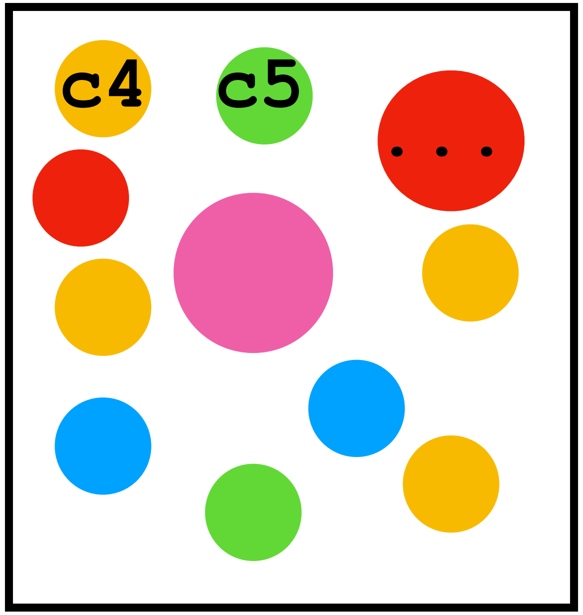
Object-Oriented Programming



```
c1.area()
```



```
c2.xpos == c3.xpos

c2.ypos == c3.ypos
```



```
getTotalArea(c4, c5, …)
```

Object-Oriented Programming

An **object** is a conceptual grouping of variables and methods that make use of those variables. ***You've been using these the entire time***

Everything in Python is an object

```
x = “myString"

print(x.capitalize())

print(x.find(“String"))

print(x.upper())

print(x[3]) # __getitem__()

print(x) # __str__()
```

Variables:

Type	String
Value	myString
Ref Count	1

Methods:

Object-Oriented Programming

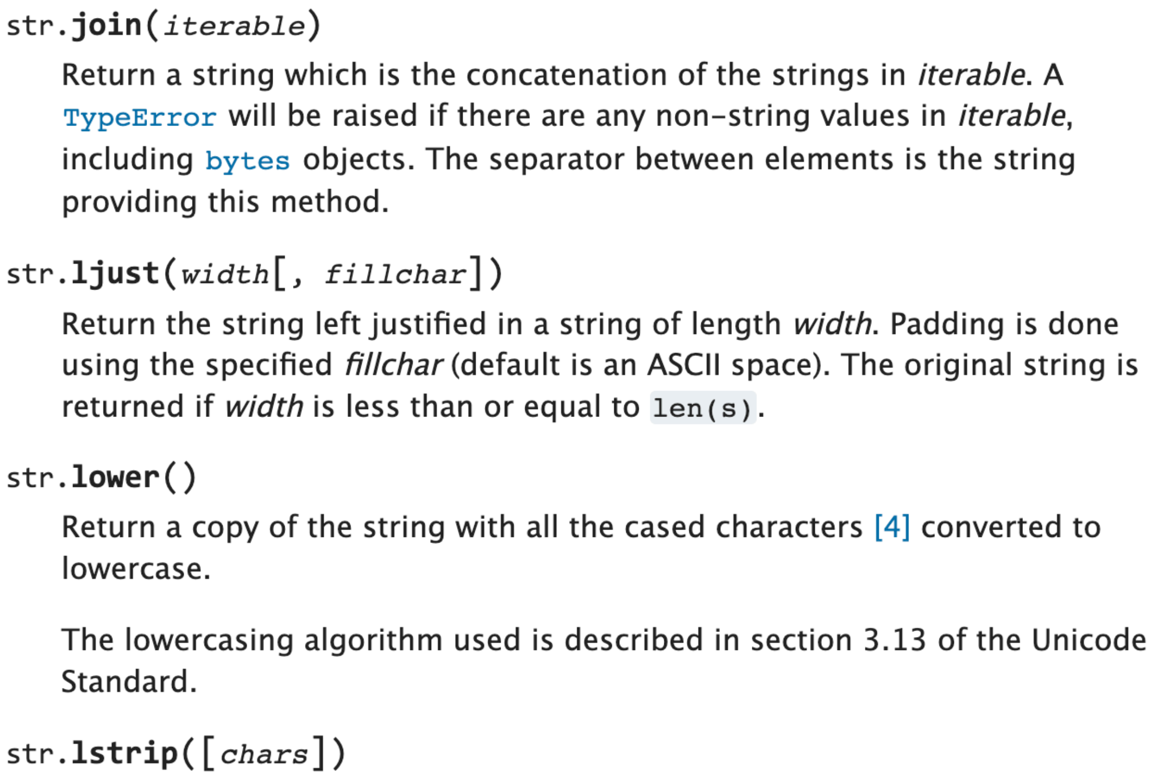
Even things that don’t have obvious function calls are (secretly) defined as a method of some object.

```
a="3"
b=3
c=3.0
d=True

print(a + b)

print(b + c)

print(c > d)
```

```
# For objects of type ‘string’
def __add__(self, o):
  ...

# For objects of type ‘int’
def __add__(self, o):
  ...

# For objects of type ‘float’
def __add__(self, o):
  ...


def __gt__(self, o):
```

Object-Oriented Programming

The collection of publicly accessible methods and variables that make up an object is its **interface.** This includes none of the implementation details.



str.**join**(*iterable*)

Return a string which is the concatenation of the strings in *iterable*. A `TypeError` will be raised if there are any non-string values in *iterable*, including `bytes` objects. The separator between elements is the string providing this method.

str.**ljust**(*width*[, *fillchar*])

Return the string left justified in a string of length *width*. Padding is done using the specified *fillchar* (default is an ASCII space). The original string is returned if *width* is less than or equal to `len(s)`.

str.**lower**()

Return a copy of the string with all the cased characters [4] converted to lowercase.

The lowercasing algorithm used is described in section 3.13 of the Unicode Standard.

str.**lstrip**([*chars*])

https://docs.python.org/3/library/stdtypes.html#string-methods

Object-Oriented Programming

We will discuss and use data structures in the context of their **interface.**

Ex: The string [data type] will have a few properties in any language

```
std::string x = "Hello World";

for(int i = x.length() - 1; i >= 0; --i){
   std::cout << x[i] << std::endl;
}
```

```
x = "Hello World"

i = len(x) - 1
while(i >= 0):
    print(x[i])
    i-=1
```

In-Class Exercise

Work with your neighbors to define an **interface** for a game of tic-tac-toe. What variables do you need? What methods would you make?

Object-Oriented Programming

The implementation details of an object is defined in a **class** definition.

You can think of this as the ‘blueprints’ to make an object.

```
PyObject * PyString_FromStringAndSize(const char *str, Py_ssize_t size)
{
    ...

    op = (PyStringObject *)PyObject_MALLOC(PyStringObject_SIZE + size);
    if (op == NULL)
        return PyErr_NoMemory();
    PyObject_INIT_VAR(op, &PyString_Type, size);
    op->ob_shash = -1;
    op->ob_sstate = SSTATE_NOT_INTERNED;
    if (str != NULL)
        Py_MEMCPY(op->ob_sval, str, size);
    op->ob_sval[size] = '\0';
    /* share short strings */
    if (size == 0) {
        PyObject *t = (PyObject *)op;
        PyString_InternInPlace(&t);
        op = (PyStringObject *)t;
        nullstring = op;
        Py_INCREF(op);
```

This is C code! Don’t worry if you cant read this

Python Class Definition

A class is defined with the keyword / syntax: ’`class <Name>:`’

```
class Circle:
    pi = 3.14

    def __init__(self,r, c, x, y):
        self.radius = r
        self.color = c
        self.xpos, self.ypos = x, y

    def __eq__(self, other):
        return (self is other)

    def circumference(self):
        return 2 * Circle.pi * self.radius

    def area(self):
        return Circle.pi * (self.radius)**2
```

Python Class Definition

A **class variable** is a variable that is shared by ALL instances of the class

```
class Circle:
    pi = 3.14

    def __init__(self,r, c, x, y):
        self.radius = r
        self.color = c
        self.xpos, self.ypos = x, y

    def __eq__(self, other):
        return (self is other)

    def circumference(self):
        return 2 * Circle.pi * self.radius

    def area(self):
        return Circle.pi * (self.radius)**2
```

Python Class Definition

A Python class constructor is defined by **’`__init__(<parameters>)`’**

```
class Circle:
    pi = 3.14

    def __init__(self, r, c, x, y):
        self.radius = r
        self.color = c
        self.xpos, self.ypos = x, y

    def __eq__(self, other):
        return (self is other)

    def circumference(self):
        return 2 * Circle.pi * self.radius

    def area(self):
        return Circle.pi * (self.radius)**2
```

Python Class Definition

Member (or instance) variables are defined with **self.<var>**.

Each object has their own *instance* of the variable with its own values.

```
class Circle:
    pi = 3.14

    def __init__(self, r, c, x, y):
        self.radius = r
        self.color = c
        self.xpos, self.ypos = x, y

    def __eq__(self, other):
        return (self is other)

    def circumference(self):
        return 2 * Circle.pi * self.radius

    def area(self):
        return Circle.pi * (self.radius)**2



```

Python Class Definition

Each object has their own *instance* of the variable with its own values.

Given a constructor, we can create new *instances* of objects.

```python
class Circle:
    pi = 3.14

    def __init__(self, r, c, x, y):
        self.radius = r
        self.color = c
        self.xpos, self.ypos = x, y
```

```python
c1 = Circle(2, "Red", 5, 5)
c2 = Circle(2, "Blue", 5, 10)
c3 = Circle(2, "Red", 5, 5)
```

Python Class Definition

It is very common to have multiple possible constructors.

Have we implemented this correctly?

```python
class Circle:
    pi = 3.14

    def __init__(self, r, x, y):
        self.radius = r
        self.color = "Black"
        self.xpos, self.ypos = x, y

    def __init__(self, r, c, x, y):
        self.radius = r
        self.color = c
        self.xpos, self.ypos = x, y


c = Circle(5, 5, 5)


c = Circle(r=5, x=5, y=5)
```

Python Class Definition

Remember our lesson from functions! Overloading uses default args.

```
class Circle:
    pi = 3.14

    def __init__(self,r, x, y, c="Black"):
        self.radius = r
        self.color = c
        self.xpos, self.ypos = x, y



c = Circle(5, 5, 5)
c = Circle(r=5, x=5, y=5)


c1 = Circle(2, 5, 5, "Red")
c2 = Circle(2, 5, 10, c="Blue")
c3 = Circle(2, 5, 5, "Red")
```

Python Class Definition

Python classes can have **member functions**.

```
class Circle:
    pi = 3.14

    def __init__(self,r, x, y, c="Black"):
        self.radius = r
        self.color = c
        self.xpos, self.ypos = x, y

    def __eq__(self, other):
        return (self is other)

    def circumference(self):
        return 2 * Circle.pi * self.radius

    def area(self):
        return Circle.pi * (self.radius)**2
```

Python Class Definition

Let’s breakdown a Python class definition:

```python
class Circle:
    pi = 3.14

    def __init__(self,r, x, y, c="Black"):
        self.radius = r
        self.color = c
        self.xpos, self.ypos = x, y

    def __eq__(self, other):
        return (self is other)

    def circumference(self):
        return 2 * Circle.pi * self.radius

    def area(self):
        return Circle.pi * (self.radius)**2



```

```python
c1 = Circle(2, "Red", 5, 5)
c2 = Circle(2, "Blue", 5, 10)
c3 = Circle(2, "Red", 5, 5)

print(c1.radius == c2.radius)

print(c1.color == c3.color)

print(c1.area())

print(c1 is c3)

print(Circle.pi)

```

Interface vs Class vs Object

Unfortunately these are not consistent terms (especially interface)

For this course:

An **interface** is the functions and operations that an object has.

“I expect to be able to print a string or add two strings together.”

A **class** is the implementation details — the code that defines all objects

“The Python string class has a method __str__() which defines print()”

An **object** is a specific instance of a class, with unique variables.

“ X and Y are two separate strings with different memory addresses and values.”

In-Class Exercise

How many unique list objects do we have in the following code?

```
x = [1, 2, 3]
y = x
z = [1, 2, 3]

print(x is y)

print(x is z)

print(x == y)

print(x == z)

print(id(x))
print(id(y))
print(id(z))

```

In-Class Exercise

Let’s return to tic-tac-toe. Can we create a Python class for a common interface?

List Abstract Data Type

What is a list? What properties does it have? What functions?

List Abstract Data Type

A list is an **ordered** collection of items

- Items can be either **heterogeneous** or **homogenous**
- The list can be of a **fixed size** or is **resizable**

List Abstract Data Type

A minimally functional list must have the following functions:

Constructor

Insert

Delete

Index

Size()**