Algorithms and Data Structures for Data Science

Object Oriented Programming

CS 277

January 23, 2023

Brad Solomon

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Department of Computer Science

Reminder: Lab_fundamentals due tonight!

No extensions for labs*

Use Piazza or office hours to get assistance.

Be sure your code is on Prairielearn and passes the autograder

Learning Objectives

Formalize and explore function overloading (in Python)

Introduce object-oriented programming

Discuss and practice defining interfaces for computational problems

Introduce class definitions in python

Python Functions

A function in Python is defined by its name and its input parameters

```
 1  def getTotalTime(checkin, checkout):
 2
 3
 4  def getSmallestEven(x, y, z):
 5
 6
 7  def electricBill(watts):
 8
 9
10  print(getTotalTime("09:00:00","17:31:53"))
11
12
13  print(getSmallestEven(2, 1, 3))
14
15
16  print(electricBill(40))
17
18
19
```

Function Overloading

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

```
 1  def combine(x, y):
 2      return [x, y]
 3
 4  print(combine(5, 1))
 5
 6  def combine(list1, list2):
 7      return list1+list2
 8
 9  print(combine([1, 2], [3, 4]))
10
11  def combine(x, list1, list2):
12      return [x]+list1+list2
13
14  print(combine(0, [1, 2], [4, 5]))
15
16
17
18
19
```

Function Overloading

To properly define an overloaded function, give default arguments.

```
 1  def combine(x, y=None, list1 = None, list2 = None):
 2      out = [x]
 3      if y:
 4          out+=[y]
 5      if list1:
 6          out+=list1
 7      if list2:
 8          out+=list2
 9      return out
10
11  print(combine(5, 1))
12
13
14  print(combine(0, [1, 2], [4, 5]))
15
16
17  print(combine(0, list1=[1, 2], list2=[4, 5]))
18
19
```

Function Overloading

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

```
 1  def combine(*args, **kwargs):
 2      out = []
 3
 4      for a in args:
 5          out.append(a)
 6
 7
 8      for k, v in kwargs.items():
 9          print("{} = {}".format(k, v))
10          out+=v
11      return out
12
13
14  print(combine(0, 1, 2, 3, 4, \
15  list1=[9, 2,3,1], list2=[8,7,2,1], \
16  list3 = [10]))
17
18
19
```

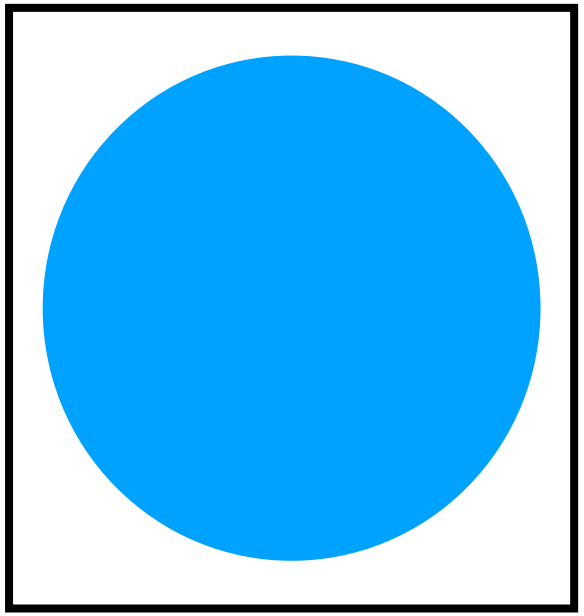
Function Overloading

You might be wondering why this is important…

```
 1  import pandas
 2
 3  pd.read_table('myFile.csv')
 4
 5
 6
 7  pd.read_table('myFile.csv',delimiter=',')
 8
 9
10
11
12  pd.read_table('myFile.csv',delimiter=',',
13  usecols = ['Netid','Grade'])
14
15
16
17
18
19
```

Object-Oriented Programming

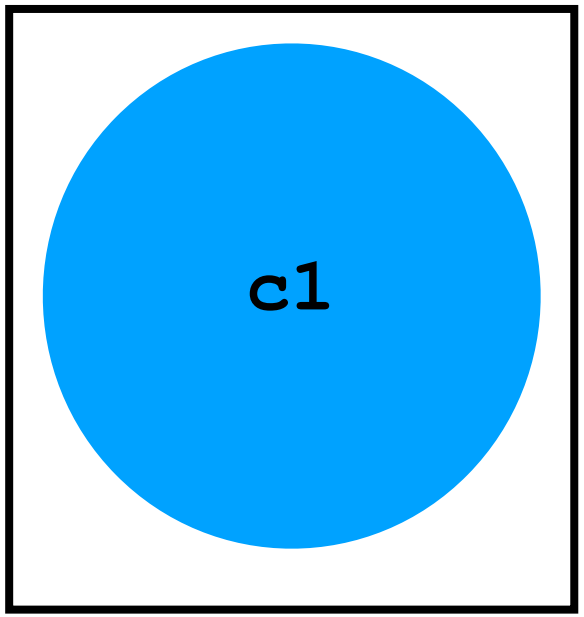
An **object** is a conceptual grouping of variables and functions that make use of those variables. A function associated with an object is a **method.**



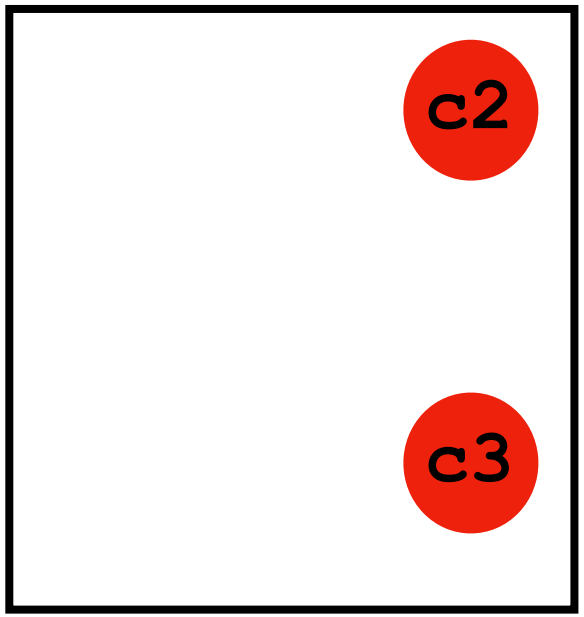
Variables:

Methods:

Object-Oriented Programming

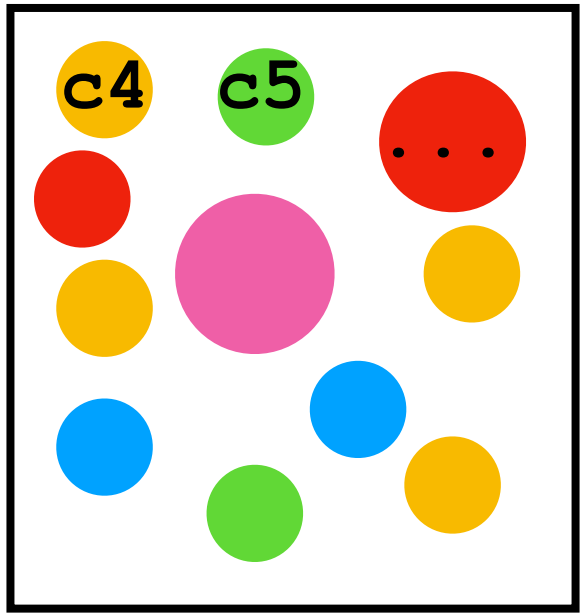


`c1.area()`



`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

```
1  x = "myString"
2
3  print(x.capitalize())
4
5  print(x.find("String"))
6
7  print(x.upper())
8
9  print(x[3]) # __getitem__()
10
11 print(x) # __str__()
12
13
14
15
16
```

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.

```
1  a="3"
2  b=3
3  c=3.0
4  d=True
5
6  print(a + b)
7
8  print(b + c)
9
10 print(c > d)
11
12
13
14
15
16
```

```
1  # For objects of type 'string'
2  def __add__(self, o):
3    ...
4
5  # For objects of type 'int'
6  def __add__(self, o):
7    ...
8
9  # For objects of type 'float'
10 def __add__(self, o):
11   ...
12
13
14 def __gt__(self, o):
15
16
```

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

```
1  std::string x = "Hello World";
2
3  for(int i = x.length() - 1; i >= 0; --i){
4     std::cout << x[i] << std::endl;
5  }
6
7
8
9
10
11
```

```
1  x = "Hello World"
2
3  i = len(x) - 1
4  while(i >= 0):
5      print(x[i])
6      i-=1
7
8
9
10
11
```

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.

```
 1  PyObject * PyString_FromStringAndSize(const char *str, Py_ssize_t size)
 2  {
 3      ...
 4
 5      op = (PyStringObject *)PyObject_MALLOC(PyStringObject_SIZE + size);
 6      if (op == NULL)
 7          return PyErr_NoMemory();
 8      PyObject_INIT_VAR(op, &PyString_Type, size);
 9      op->ob_shash = -1;
10      op->ob_sstate = SSTATE_NOT_INTERNED;
11      if (str != NULL)
12          Py_MEMCPY(op->ob_sval, str, size);
13      op->ob_sval[size] = '\0';
14      /* share short strings */
15      if (size == 0) {
16          PyObject *t = (PyObject *)op;
17          PyString_InternInPlace(&t);
18          op = (PyStringObject *)t;
19          nullstring = op;
20          Py_INCREF(op);
```

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

Python Class Definition

Let's breakdown a Python class definition:

```
 1  class Circle:
 2      pi = 3.14
 3
 4      def __init__(self,r, c, x, y):
 5          self.radius = r
 6          self.color = c
 7          self.xpos, self.ypos = x, y
 8
 9      def __eq__(self, other):
10          return (self is other)
11
12      def circumference(self):
13          return 2 * Circle.pi * self.radius
14
15      def area(self):
16          return Circle.pi * (self.radius)**2
17
18
19
```

Python Class Definition

Let's breakdown a Python class definition:

```
 1  class Circle:
 2      pi = 3.14
 3
 4      def __init__(self,r, c, x, y):
 5          self.radius = r
 6          self.color = c
 7          self.xpos, self.ypos = x, y
 8
 9      def __eq__(self, other):
10          return (self is other)
11
12      def circumference(self):
13          return 2 * Circle.pi * self.radius
14
15      def area(self):
16          return Circle.pi * (self.radius)**2
17
18
19
```

Python Class Definition

Let's breakdown a Python class definition:

```
 1  class Circle:
 2      pi = 3.14
 3
 4      def __init__(self, r, c, x, y):
 5          self.radius = r
 6          self.color = c
 7          self.xpos, self.ypos = x, y
 8
 9      def __eq__(self, other):
10          return (self is other)
11
12      def circumference(self):
13          return 2 * Circle.pi * self.radius
14
15      def area(self):
16          return Circle.pi * (self.radius)**2
17
18
19
```

Python Class Definition

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

```
1  class Circle:
2      pi = 3.14
3
4      def __init__(self, r, c, x, y):
5          self.radius = r
6          self.color = c
7          self.xpos, self.ypos = x, y
```

```
21 c1 = Circle(2, "Red", 5, 5)
22 c2 = Circle(2, "Blue", 5, 10)
23 c3 = Circle(2, "Red", 5, 5)
24
25
26
27
28
29
30
31
```

Python Class Definition

Let's breakdown a Python class definition:

```
 1  class Circle:
 2      pi = 3.14
 3
 4      def __init__(self, r, x, y):
 5          self.radius = r
 6          self.color = "Black"
 7          self.xpos, self.ypos = x, y
 8
 9      def __init__(self, r, c, x, y):
10          self.radius = r
11          self.color = c
12          self.xpos, self.ypos = x, y
13
14
15  c = Circle(5, 5, 5)
16
17
18  c = Circle(r=5, x=5, y=5)
19
```

Python Class Definition

Let's breakdown a Python class definition:

```
1  class Circle:
2      pi = 3.14
3
4      def __init__(self,r, x, y, c="Black"):
5          self.radius = r
6          self.color = c
7          self.xpos, self.ypos = x, y
8
9
10
11 c = Circle(5, 5, 5)
12 c = Circle(r=5, x=5, y=5)
13
14
15 c1 = Circle(2, 5, 5, "Red")
16 c2 = Circle(2, 5, 10, c="Blue")
17 c3 = Circle(2, 5, 5, "Red")
18
19
```

Python Class Definition

Let's breakdown a Python class definition:

```
 1  class Circle:
 2      pi = 3.14
 3
 4      def __init__(self,r, x, y, c="Black"):
 5          self.radius = r
 6          self.color = c
 7          self.xpos, self.ypos = x, y
 8
 9      def __eq__(self, other):
10          return (self is other)
11
12      def circumference(self):
13          return 2 * Circle.pi * self.radius
14
15      def area(self):
16          return Circle.pi * (self.radius)**2
17
18
19
```

Python Class Definition

Let's breakdown a Python class definition:

```
 1  class Circle:
 2      pi = 3.14
 3
 4      def __init__(self,r, x, y, c="Black"):
 5          self.radius = r
 6          self.color = c
 7          self.xpos, self.ypos = x, y
 8
 9      def __eq__(self, other):
10          return (self is other)
11
12      def circumference(self):
13          return 2 * Circle.pi * self.radius
14
15      def area(self):
16          return Circle.pi * (self.radius)**2
17
18
19
```

```
21  c1 = Circle(2, "Red", 5, 5)
22  c2 = Circle(2, "Blue", 5, 10)
23  c3 = Circle(2, "Red", 5, 5)
24
25  print(c1.radius == c2.radius)
26
27  print(c1.color == c3.color)
28
29  print(c1.area())
30
31  print(c1 is c3)
32
33  print(Circle.pi)
34
```

Interface vs Class vs Object

Unfortunately these are not consistent terms (especially interface)

For this class:

An **interface** is the expected input / output of an object

"A string will always have a length though how I access it might change"

A **class** is the implementation details / coding of the interface

"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 instances of a list do we have in the following code?

```
 1  x = [1, 2, 3]
 2  y = x
 3  z = [1, 2, 3]
 4
 5  print(x is y)
 6
 7  print(x is z)
 8
 9  print(x == y)
10
11  print(x == z)
12
13  print(id(x))
14  print(id(y))
15  print(id(z))
16
```

In-Class Exercise

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