

Algorithms and Data Structures for Data Science

Functions and Objects

CS 277

January 23, 2024

Brad Solomon



UNIVERSITY OF
ILLINOIS
URBANA - CHAMPAIGN

Department of Computer Science

CS 277 should be low-stress medium workload

If you are struggling to complete assignments, ask for help!

1. Attend office hours (see schedule on website)
2. Email professor (Include CS 277 in subject heading)
3. Talk before or after class
4. Ask questions online through Piazza or Discord

Course Discord Link on Prairielearn

Current link invite valid for 7 days

Strongly encouraged to join before link invalidates

Lab / Course Feedback

Feedback is necessary to keep course pacing appropriate for all

Online Asynchronous Options:

Discord, Piazza, Email, Feedback Forms

In-Person:

In-class questions, labs, office hours

This is especially important in the early stages of the class.

Learning Objectives

Continue reviewing Python Fundamentals

Continue building a programming pipeline

Discuss and practice defining interfaces for computational problems

Programming Toolbox: Data Type Casting

Variables in Python are **strongly typed** and **dynamically typed**

```
1 x = 1.1
2 y = "3"
3 z = "4x"
4
5 print(int(x))
6
7 print(float(y))
8
9 print(int(z))
10
11 a = True
12 b = 5
13
14 print(a - a)
15
16 print(int(a))
17
18 print(bool(b))
```

Programming Pipeline Part 2

1. Make sure you understand the problem

What is the **input** and **output** of the problem?

Can you break the problem down into parts?

Do any of the sub-problems build off each other?

2. Solve (and test) each part one at a time

What should the output be given an input?

Are there any edge cases you are missing?

Debugging your Code (PrairieLearn)

“I submitted my code and didnt get points. Now what?”

× **[0/5]** Check 277student() Random Tests



× **[0/5]** Check checkSorted() Tests



× **[0/5]** Check geqThan() Random Tests



× **[0/5]** Check getGrade() Tests



Max points: 5

Earned points: 0

Message

- Grading was skipped because an earlier test failed -

Debugging your Code (PrairieLearn)

Autograder is designed to give feedback on what went wrong!

× **[0/5]** Check `geqThan()` Random Tests ^

Max points: 5

Earned points: 0

Message

```
The callable 'geqThan' supplied in your code failed with an exception while it was being called.  
File "/grade/run/code_feedback.py", line 448, in call_user  
    return f(*args, **kwargs)  
TypeError: geqThan() missing 1 required positional argument: 'boundary'
```

Debugging your Code (PrairieLearn)

getSmallestEven() has the following return message:

'Test: 55, 84, 27' is None or not defined

electricBill() has the following return message:

'Test: 505' looks good

'Test: 49' looks good

'Test: 477' is inaccurate

Python Toolbox: Print Statements

```
print("ABC"+"DEF"+str(x))
```

```
print(f"Hello {x}, its nice to meet you!")
```

```
print("{} , {} , {}".format(1, 2, 3))
```

Python Toolbox: Print Statements



```
1 def buildString(inList):
2     i=""
3
4     for i in inList:
5         i+=i
6
7     return i
8
```

Programming Toolbox: Functions

Functions are defined by `def <name>(<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
```

Programming Toolbox: Functions

Functions in Python are everywhere!

```
1 def f1(x, y):  
2     z = x + y  
3     return z  
4  
5 print(f1(1, 3))  
6  
7  
8 print(f1([0, 1, 2], [3, 4, 5]))  
9  
10  
11
```

```
operator.add(a, b) ¶  
operator.__add__(a, b)  
Return a + b, for a and b numbers.
```

`__add__(a, b)` also works for lists!

```
1 a = True  
2 b = 5  
3  
4 print(a - a)  
5  
6  
7
```

What did this return? Why?

Python Toolbox: Functions

What does it mean to be the **‘building block of programming’**?

Python is built on objects, objects are [partially] defined by functions

```
1 x = "string"
2
3 y = x.upper()
4
5 print(y)
6 print(y.lower())
7
8
```

`str.upper()`

Return a copy of the string with all the cased characters [4] converted to **uppercase**. Note that `s.upper().isupper()` might be **False** if `s` contains uncased characters or if the Unicode category of the resulting character(s) is not “Lu” (Letter, uppercase), but e.g. “Lt” (Letter, titlecase).

The uppercasing algorithm used is [described in section 3.13 ‘Default Case Folding’ of the Unicode Standard](#).

Python Toolbox: Functions

Learning how to read a function description is essential!

```
str.split(sep=None, maxsplit=- 1)
```

Return a list of the words in the string, using *sep* as the delimiter string. If *maxsplit* is given, at most *maxsplit* splits are done (thus, the list will have at most `maxsplit+1` elements). If *maxsplit* is not specified or `-1`, then there is no limit on the number of splits (all possible splits are made).

If *sep* is given, consecutive delimiters are not grouped together and are deemed to delimit empty strings (for example, `'1,,2'.split(',')` returns `['1', '', '2']`). The *sep* argument may consist of multiple characters (for example, `'1<>2<>3'.split('<>')` returns `['1', '2', '3']`). Splitting an empty string with a specified separator returns `['']`.

For example:

```
>>> '1,2,3'.split(',')
['1', '2', '3']
>>> '1,2,3'.split(',', maxsplit=1)
['1', '2,3']
>>> '1,2,,3,.'.split(',')
['1', '2', '', '3', '']
```

```
>>>
```


Python Toolbox: Functions

Learning how to read a function description is essential!

```
str.split(sep=None, maxsplit=- 1)
```

Return a list of the words in the string, using most *maxsplit* splits are done (thus, the list will have at most *maxsplit* + 1 elements). If *maxsplit* is not specified or `-1`, then there is no limit on the number of splits (i.e., all possible splits are made).

If *sep* is given, consecutive delimiters are not grouped together and are recognized as empty delimiters (for example, `'1,,2'.split(',')` results in `['1', '', '2']`). If *sep* consists of multiple characters, they are all used as delimiters (for example, `'1,2,3'.split(' ,')` results in `['1', '2', '3']`). Splitting an empty string with a specified *sep* returns `['']`.

For example:

```
>>> '1,2,3'.split(',')
['1', '2', '3']
>>> '1,2,3'.split(',', maxsplit=1)
['1', '2,3']
>>> '1,2,,3,.'.split(',')
['1', '2', '', '3', '']
```

When in doubt — read the docs!



<https://docs.python.org/3.12/>

Your favorite search engine can also go a long way!

Lets practice — what does the string **strip()** function do?

Programming Practice: Functions

It is also important to be able to read a function given code

```
1 # INPUT: None
2 # OUTPUT: None
3 def f1():
4     print('Function A called')
5
6 # INPUT: A Python object
7 # OUTPUT: The same Python object unchanged
8 def f2(input):
9     print("Function B called")
10    return input
11
12 # INPUT: A function that accepts zero args
13 # OUTPUT: The return value of the function
14 def f3(input):
15     print("Function C called")
16     return input()
17
18
19
```

```
print(f1())
```

```
print(f2(5)+3)
```

```
print(f2("Hi")+ " Bye")
```

Programming Practice: Functions

What gets printed when running the following function call?

```
1 # INPUT: None
2 # OUTPUT: None
3 def f1():
4     print('Function A called')
5
6 # INPUT: A Python object
7 # OUTPUT: The same Python object unchanged
8 def f2(input):
9     print("Function B called")
10    return input
11
12 # INPUT: A function that accepts zero args
13 # OUTPUT: The return value of the function
14 def f3(input):
15     print("Function C called")
16     return input()
17
18
19
```

```
print(f3(f2(f1)))
```

Programming Practice: Functions

Each function is its own 'frame' or 'scope'

```
1 # INPUT: None
2 # OUTPUT: None
3 def f1():
4     print('Function A called')
5
6 # INPUT: A Python object
7 # OUTPUT: The same Python object unchanged
8 def f2(input):
9     print("Function B called")
10    return input
11
12 # INPUT: A function that accepts zero args
13 # OUTPUT: The return value of the function
14 def f3(input):
15     print("Function C called")
16     return input()
17
18 print(f3(f2(f1)))
19
```

Programming Practice: Functions

What happens when running the following function calls?

```
1 # INPUT: None
2 # OUTPUT: None
3 def f1():
4     print('Function A called')
5
6 # INPUT: A Python object
7 # OUTPUT: The same Python object unchanged
8 def f2(input):
9     print("Function B called")
10    return input
11
12 # INPUT: A function that accepts zero args
13 # OUTPUT: The return value of the function
14 def f3(input):
15     print("Function C called")
16     return input()
17
18
19
```

`print(f2(f1))`

`print(f3)`

Programming Practice: Functions



What will the functions here print?

```
1 def increase(ival):
2     ival+=1
3     return ival
4
5 def doubleInc(ival):
6     y = increase(ival)
7     y += increase(ival)
8     return y
9
10 print(increase(5)) # should return 6
11
12
13
14
15 print(doubleInc(7)) # should return 9
16
17
18
19
```

Programming Practice: Function Scope

```
1 def f1(x, y):
2     x = x + y
3     return x
4
5 def f2(z):
6     z = [0]
7
8 def f3(z):
9     z[0]=4
10
11 print(x)
12
13 a, b = 2, 5
14 print(f1(a, b))
15 print(a, b)
16
17 test = [0, 1, 2]
18 f2(test)
19 print(test)
20
21 f3(test)
22 print(test)
23
```

Each frame has its own variables.

Programming Practice: Function Scope

```
1 def f1(x, y):  
2     x = x + y  
3     return x  
4  
5 def f2(z):  
6     z = [0]  
7  
8 def f3(z):  
9     z[0]=4  
10  
11 print(x)  
12
```

Each frame has its own variables.

Global

F1()

F2()

F3()

NameError

Traceback (most recent call last)

[/Users/bradsol/Desktop/UIUC/cs277/website/assets/code/sp24/funcI0_public.ipynb](#) Cell 13 line 1

```
14 def f3(z):  
15     z[0]=4  
--> 17 print(x)  
19 a, b = 2, 5  
20 print(f1(a, b))
```

NameError: name 'x' is not defined

Programming Practice: Function Scope



Each frame has its own variables.

```
1 def f1(x, y):
2     x = x + y
3     return x
4
5 def f2(z):
6     z = [0]
7
8 def f3(z):
9     z[0]=4
10
11
12
13 a, b = 2, 5
14 print(f1(a, b))
15 print(a, b)
16
17 test = [0, 1, 2]
18 f2(test)
19 print(test)
20
21 f3(test)
22 print(test)
23
```

Global

F1()

F2()

F3()

Python Toolbox: Functions

Many built-in functions can take a variety of input arguments

```
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=',', usecols = ['Netid','Grade'])
13
14
15
16
17
18
19
```

Programming Toolbox: 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
```

Programming Toolbox: 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
```

Programming Toolbox: Function Overloading



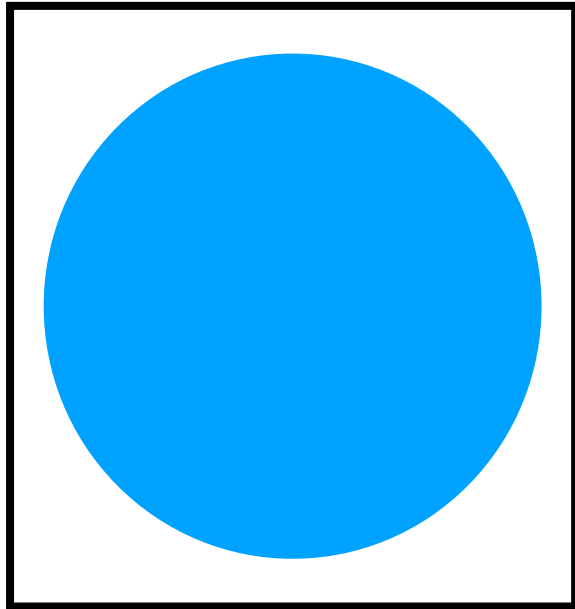
For true freedom of input, use keyword `*args` and `**kwargs`

```
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
```

Object-Oriented Programming

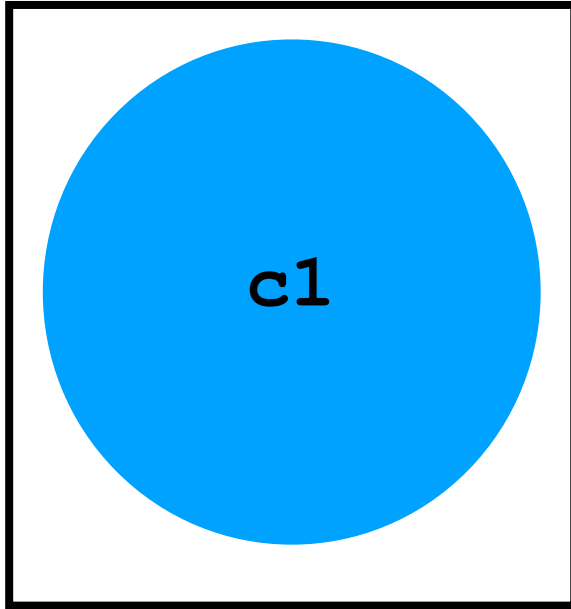
An **object** is a conceptual grouping of variables and functions that make use of those variables. A function associated with an object is a **method**.

Variables:

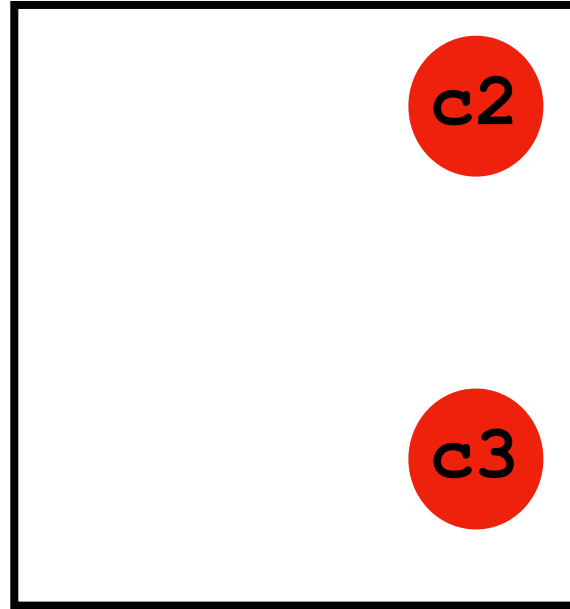


Methods:

Object-Oriented Programming

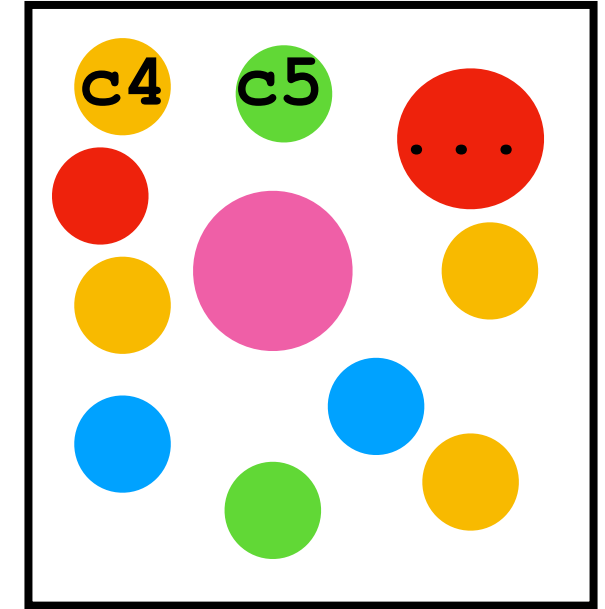


`c1.area()`



`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 **Variables:**

```
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
```

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?