Functions and Objects
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

6 days?

Strongly encouraged to join before link invalidates

Lecture channel on Discord
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
Variables in Python are **strongly typed** and **dynamically typed**
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 didn't get points. Now what?”

<table>
<thead>
<tr>
<th>0/5</th>
<th>Check 277studen() Random Tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>0/5</td>
<td>Check checkSorted() Tests</td>
</tr>
<tr>
<td>0/5</td>
<td>Check geqThan() Random Tests</td>
</tr>
<tr>
<td>0/5</td>
<td>Check getGrade() Tests</td>
</tr>
</tbody>
</table>

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!

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

⇒ No return

electricBill() has the following return message:

'Test: 505' looks good
'Test: 49' looks good
'Test: 477' is inaccurate

⇒ why did 477 fail?

1) What should eB(477) output?
2) What is your code outputting?
Python Toolbox: Print Statements

```
x = 5
print("ABC" + "DEF" + str(x))  
    # String formatting
    # Add (plus) +
    # A + B + 5
    # Easy to use for simple strings

print(f"Hello {x}, its nice to meet you!")  
    # Variable inside the f

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

```
"1, 2, 3"
```

```
X + "", + Y + "," + Z
```

```
"1, 2, 3"
```

```
X + "," + Y + "," + Z
```

```
def buildString(inList):
    i=""
    for i in inList:
        i+=i
    return i
```
Programming Toolbox: Functions

Functions are defined by `def <name>(<parameters>):`

```python
def getTotalTime(checkin, checkout):
    return

def getSmallestEven(x, y, z):
    pass

def electricBill(watts):
    pass

print(getTotalTime("09:00:00","17:31:53"))
print(getSmallestEven(2, 1, 3))
print(electricBill(40))
```
Programming Toolbox: Functions

Functions in Python are everywhere!

def f1(x, y):
    z = x + y
    return z

print(f1(1, 3))
print(f1([0, 1, 2], [3, 4, 5]))

operator.__add__(a, b)  # __add__(a, b) also works for lists!

a = True
b = 5
print(a - a)
print(a - b)

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

```python
x = "string"
y = x.upper()
print(y)
print(y.lower())
```

**str.upper()**

Return a copy of the string with all the cased characters 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!

```python
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:

```python
>>> '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!

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

Return a list of the words in the string, using `sep` as the delimiter string.

- The most `maxsplit` splits are done (thus, the list returned contains no more than `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 considered as a single delimiter that splits at them, and the resulting lists are joined.

Strings that consist of multiple characters (for example, `'1,,2'.split(',')`), or consist of multiple characters and a non-separator (for example, `'1,, ,2'.split(',')`) are split.

Splitting an empty string with a specified separator is allowed.

For example:

```python
>>> '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?

- `strip()` removes whitespace from start and end
- `strip('A')` removes any whitespace before and after the string "A"
It is also important to be able to read a function given code.

```python
# INPUT: None
# OUTPUT: None
def f1():
    print('Function A called')

# INPUT: A Python object
# OUTPUT: The same Python object unchanged
def f2(input):
    print('Function B called')
    return input

# INPUT: A function that accepts zero args
# OUTPUT: The return value of the function
def f3(input):
    print('Function C called')
    return input()
```

```
print(f1())  # Output: None

print(f2(5)+3)  # Output: 8

print(f2("Hi")+" Bye")  # Output: "Hi Bye"
```
What gets printed when running the following function call?

```
# INPUT: None
# OUTPUT: None
def f1():
    print('Function A called')

# INPUT: A Python object
# OUTPUT: The same Python object unchanged
def f2(input):
    print('Function B called')
    return input

# INPUT: A function that accepts zero args
# OUTPUT: The return value of the function
def f3(input):
    print('Function C called')
    return input()

print(f3(f2(f1)))
```
Programming Practice: Functions

Each function is its own ‘frame’ or ‘scope’

```python
# INPUT: None
# OUTPUT: None
def f1():
    print('Function A called')

# INPUT: A Python object
# OUTPUT: The same Python object unchanged
def f2(input):
    print("Function B called")
    return input

# INPUT: A function that accepts zero args
# OUTPUT: The return value of the function
def f3(input):
    print("Function C called")
    return input()

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

Each function is its own ‘frame’ or ‘scope’
# Programming Practice: Functions

What happens when running the following function calls?

```python
print(f2(f1))
print(f3)
```
def increase(inval):
inval+=1
return inval

def doubleInc(inval):
y = increase(inval)
y += increase(inval)
return y

print(increase(5)) # should return 6
print(doubleInc(7)) # should return 9
Programming Practice: Function Scope

Each frame has its own variables.
Programming Practice: Function Scope

```
def f1(x, y):
    x = x + y
    return x

def f2(z):
    z = [0]

def f3(z):
    z[0] = 4

print(x)

a, b = 2, 5
print(f1(a, b))
print(a, b)
test = [0, 1, 2]
f2(test)
print(test)
f3(test)
print(test)
```

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/func10_public.ipynb Cell 13 line 1
```
def f3(z):
    z[0] = 4
    --> print(x)
    a, b = 2, 5
    print(f1(a, b))
```

NameError: name 'x' is not defined
Each frame has its own variables.
Python Toolbox: Functions

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

```python
import pandas

pd.read_table('myFile.csv')

pd.read_table('myFile.csv', delimiter=',' )

pd.read_table('myFile.csv', delimiter=',' , usecols=['Netid', 'Grade'] )
```
Two functions are **overloaded** when they have the same name but different parameters.

```python
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]))
```
To properly define an overloaded function, give default arguments.

```python
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

```python
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]))
```
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, ...)

c1
c2

c3
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**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>x = “myString”</code></td>
</tr>
<tr>
<td>2</td>
<td><code>print(x.capitalize())</code></td>
</tr>
<tr>
<td>3</td>
<td><code>print(x.find(“String”))</code></td>
</tr>
<tr>
<td>4</td>
<td><code>print(x.upper())</code></td>
</tr>
<tr>
<td>5</td>
<td><code>print(x[3])</code> # <code>__getitem__()</code></td>
</tr>
<tr>
<td>6</td>
<td><code>print(x)</code> # <code>__str__()</code></td>
</tr>
</tbody>
</table>

**Variables:**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>String</td>
</tr>
<tr>
<td>Value</td>
<td>myString</td>
</tr>
<tr>
<td>Ref Count</td>
<td>1</td>
</tr>
</tbody>
</table>

**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)
```

```
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 def __gt__(self, o):
14     ...
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.

```python
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])
```

[4]: 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

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

```python
x = "Hello World"
for i in range(len(x) - 1, -1, -1):
    print(x[i])
```
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?