CS 1: Intro to CS

More on Strings and Variables/Scoping

Mon. April 8th, 2024

Announcements

Reminder that HW1 is due Thursday 11:30PM on CodePost

Come to Office Hours! Even just to say hello to TAs this week :)

Tomorrow: Lab 01

- Pitfalls and debugging activity
- Lecture 3 slides include walkthrough of using the VSCode debugger El demo'd on Friday (you'll get more practice using this in lab)

Today's Learning Objectives

Deeper into strings

- Common string methods
- Indexing
- More on formatting

Get more practice with functions/scoping (live-coding exercises continuing off of Lecture 3's lec03.py function)

Documenting our code with docstrings: How and why

A bit of PEP8 style guidelines along the way (official guide here)

Check Your Understanding

Suppose we're trying to produce the following output:

```
Meet the 3 stooges!
Curly and Larry and Moe
```

What is wrong with the following code which attempts to produce this?

```
1 cat1 = Curly
2 cat2 = Larry
3 cat3 = Moe
4
5 print('Meet the 3 stooges!')
6 print(cat1 and cat2 and cat3)
```

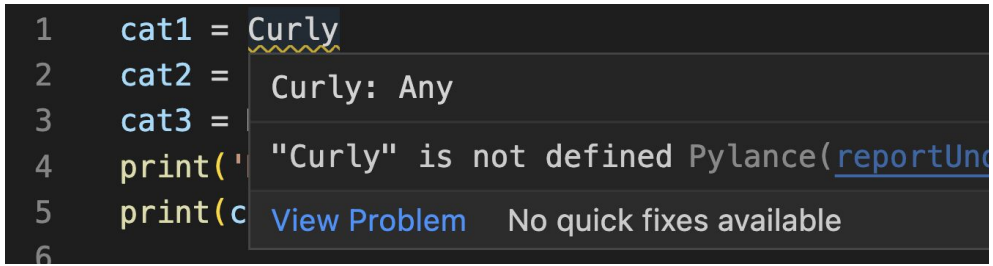
VSCode Hints

VSCode will provide "hints" when you have a .py file open

These are very useful as you're learning programming and the rules of Python

Moving your cursor over the first yellow-underline gives a message that **`Curly`** is being referenced in the RHS of an assignment (expecting an expression) but there is no variable called **`Curly`**; don't forget to distinguish between variables (no quotes) and strings (quoted) in Python!

```
1    cat1 = Curly
2    cat2 = Larry
3    cat3 = Moe
4    print('Meet the 3 stooges!')
5    print(cat1 and cat2 and cat3)
```



VSCode Hints

A caveat: not all hints will be useful, and not all bugs will be found for you in VSCode

In the example below, we've fixed the first lines, but there's still a bug!

. . . ???

More on Format Strings

Sometimes, we want to format numbers in a specific way. Here, `:d` formats integers, `:f` formats floats, and `:.2f` formats a float with exactly 2 digits following the decimal.

```
'an integer: {:d}'.format(42)
>>> 'an integer: 42'
'a float: {:f}'.format(42.123400)
>>> 'a float: 42.123400'
'a float: {:.2f}'.format(42.123400)
>>> 'a float: 42.12'
>>> 'If your hourly salary is ${:.2f}, you earn ${:.2f} for working 25 hours a
week.'.format(salary, weekly_salary)
'If your hourly salary is $18.50, you earn $462.50 for working 25 hours a week.'
```

More on Format Strings (Version 3, Preferred)

Using **`.format()`** in format strings can be tedious

```
'x = {}, b = {}'.format(a, b)'
```

There is conveniently a shortcut:

```
f'a {a}, b = {b}'
```

The **`f'...'`** syntax indicates that it's a format string, and allows you to use variables in the format string without the **`.format()`** method call

You can also add modifiers (e.g. **`{a:.2f}`**) to substitute the numeric value of a to 2 decimal points

String Indexing

Can access parts of string sequences in various ways

```
>>> name = 'Bowie'
>>> name[0]
'B'
>>> name[-1]
'e'
>>> name[len(name) - 1]
'e'
>>> name[len(name)]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Python uses “0-based” indexing, meaning the first character is at index 0, not 1

We’ll learn more about string-processing next week

Recall: Functions

A function takes some input data and transforms it into output data

Functions must be defined and then called with the appropriate arguments

A few functions are built-in to Python so we don't have to define them ourselves:

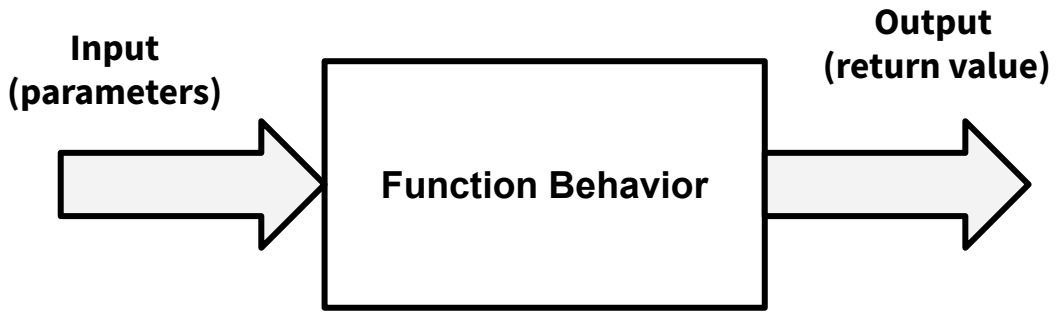
- `print(x)`
- `input(x)`
- `type(x)`
- `int(x), float(x), str(x)`
- `min(x, y, ...), max(x, y, ...)`
- `help(), help(fn)`
- `...`

Anatomy of a Function

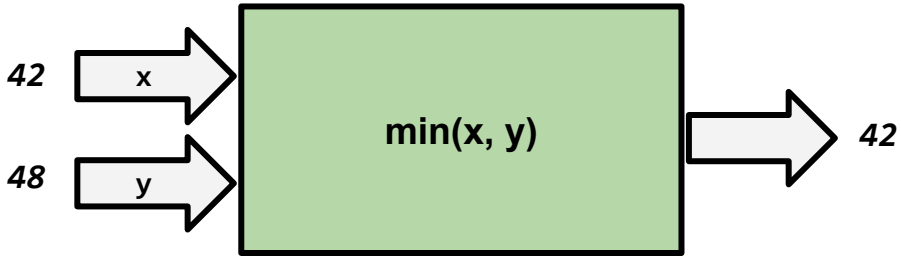
A function is like a machine to perform tasks and possibly return some result

Every function has:

- Behavior (body)
- Parameters (optional)
- Return value (optional)



Example with built-in `min` function:



Defining and Calling Functions

Functions may have parameters passed to help generalize functionality and may also specify a return value with the return keyword (**None** if no return specified)

Definition Syntax:

```
def name(<parameters>):
    <body>
    return <value> # optional
```

Definition Examples:

```
def say_hello(name):
    print('Hello ' + name + '!')

def f(x, y):
    return x + 2 * y
```

Function Call Examples:

```
say_hello('world')   # Hello world!
say_hello('Caltech') # Hello Caltech!
ans = f(2, 20)       # ans == 42
```

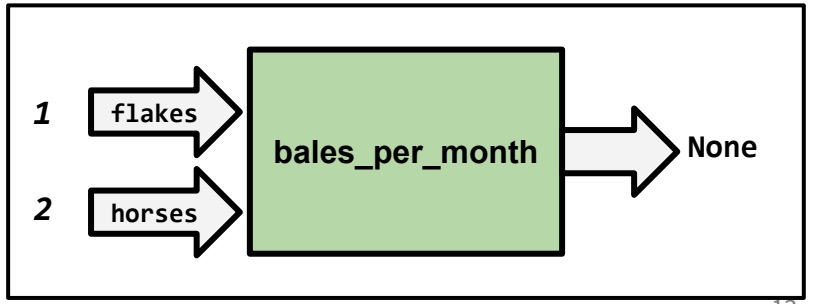
From Friday

```
def bales_per_month(flakes, horses):
    """
    Given the number of flakes of hay eaten per horse per
    day and the number of horses, reports the number
    of bales of hay needed for a month.
    """
    bales = 16 # number of flakes in a bale
    # number of flakes eaten per day
    flakes_per_day = flakes * horses

    # TODO: Figure out how to use datetime library
    number_of_days_in_month = 30
    flakes_per_month = flakes_per_day * number_of_days_in_month
    # number of bales needed for the month
    total = flakes_per_month / bales
    print(f'The number of bales needed is {total}')


# Examples calling function
bales_per_month(1, 2)
bales_per_month(3, 4)
```

Review: In practice, functions more commonly *return* a computed value instead of print it

Why?

What do we need to change here? How could we factor the printing outside of the function definition?



From Friday

```
def bales_per_month(flakes, horses):
    """
    Given the number of flakes of hay eaten per horse per
    day and the number of horses, returns the number
    of bales of hay needed for a month.
    """
    bales = 16 # number of flakes in a bale
    # number of flakes eaten per day
    flakes_per_day = flakes * horses

    # TODO: Figure out how to use datetime library
    number_of_days_in_month = 30  # For now, hard-code for April
    flakes_per_month = flakes_per_day * number_of_days_in_month
    # number of bales needed for the month
    total = flakes_per_month / bales
    print(f'The number of bales needed is {total}')


# Examples calling function
bales1 = bales_per_month(1, 2)
bales2 = bales_per_month(3, 4)
print(f'The first number of bales needed is {bales1}')
print(f'The second number of bales needed is {bales2}')
```

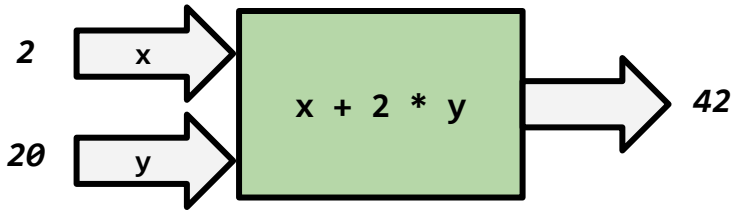
Review: In practice, functions more commonly *return* a computed value instead of print it

Why?

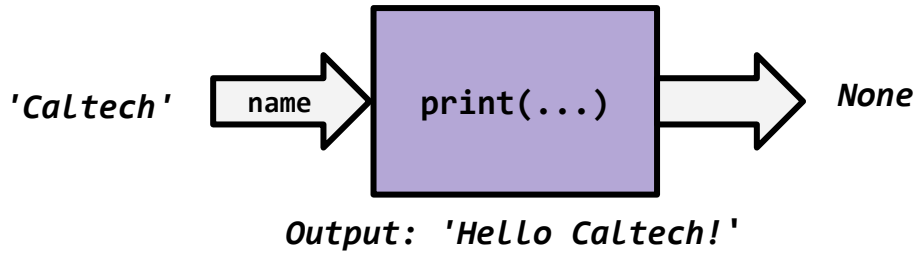
What do we need to change here? How could we factor the printing outside of the function definition?

Functions as Machines

```
1 # Defining the function
2 def f(x, y):
3     return x + 2 * y
4
5 # Calling the function
6 ans = f(2, 20)
```




```
1 # Defining the function
2 def say_hello(name):
3    print('Hello', name, '!')
4
5 # Calling the function
6 say_hello('Caltech')
```



Scope is Important!

So far, our variables have been defined top-down - later assignments will **shadow** earlier ones.

Functions introduce their own **local** scope - **variables inside functions only exist in during the lifetime of a function call**.

(Code demo with lec04_starter.zip)

Local Variables

```
1  def f(x, y):
2      z = 2 * y # z is a local variable
3      return z + x
4
5  ans1 = f(2, 20) # 42
6  ans2 = f(x, 20) # error! x is not in scope here
```

Parameters vs. Arguments

Formal parameters are simply names for the argument values passed in a function call. The **position of arguments** will determine what formal parameter name they are assigned.

They have no relationship to other variable names in the program and will override other variables if there is a naming conflict.

```
1  def f(x, y):
2      z = 2 * y # here, z is a local variable!
3      return z + x
4
5  a = 2
6  b = 20
7  ans1 = f(a, b)
8  ans2 = f(b, a) # b and a are mapped to x and y in f, respectively
```

Practice: Variables and Scoping

What is the result of executing the following program?

```
1  x = 1
2  y = 2
3  z = 3
4
5  def square(x):
6      return x * x
7
8  def mystery(x, y, z):
9      print('x: ', x, 'y: ', y, 'z: ', z)
10
11 mystery(x, y, z)
12 mystery(x + y, x, square(y))
```

Practice: Variables and Scoping

What is the result of executing the following program?

```
1  x = 1
2  y = 2
3  z = 3
4
5  def square(x):
6      return x * x
7
8  def mystery(x, y, z):
9      print('x: ', x, 'y: ', y, 'z: ', z)
10
11 mystery(x, y, z)
12 mystery(x + y, x, square(y))
```

```
Output:
x: 1 y: 2 z: 3
x: 3 y: 1 z: 4
```

Practice: String Functions

Problem: Suppose Caltech usernames were automatically generated in the format of first initial followed by full last name (making an unrealistic assumption that everyone has a single first and last name and there are no duplicates). For example, “Lorem Hovik” would generate “lhovik@caltech.edu).

Write a function called **`generate_username`** that helps generate usernames for new Caltech students. Then, add a **`generate_email`** function that uses this function to generate a caltech email address (similar to Lab01's warmup) given a first name and last name.

```
username_to_email('Lhovik')           # returns 'lhovik@caltech.edu'
generate_username('Lorem', 'Hovik')   # returns 'lhovik'
generate_email('Lorem', 'Hovik')      # returns 'lhovik@caltech.edu'
```

Comments (from Week 1)

Comments are lines in the source code that are notes to the reader(s), while Python just ignores them

Comments start with # and continue to the end of the line

```
# U.S. dollars per hour
salary = 18.5 # everything after the comment symbol is ignored
```

We'll learn other ways to document your code properly ~~next week~~ **today**!

Docstrings

Comments are commonly used to describe what a function does:

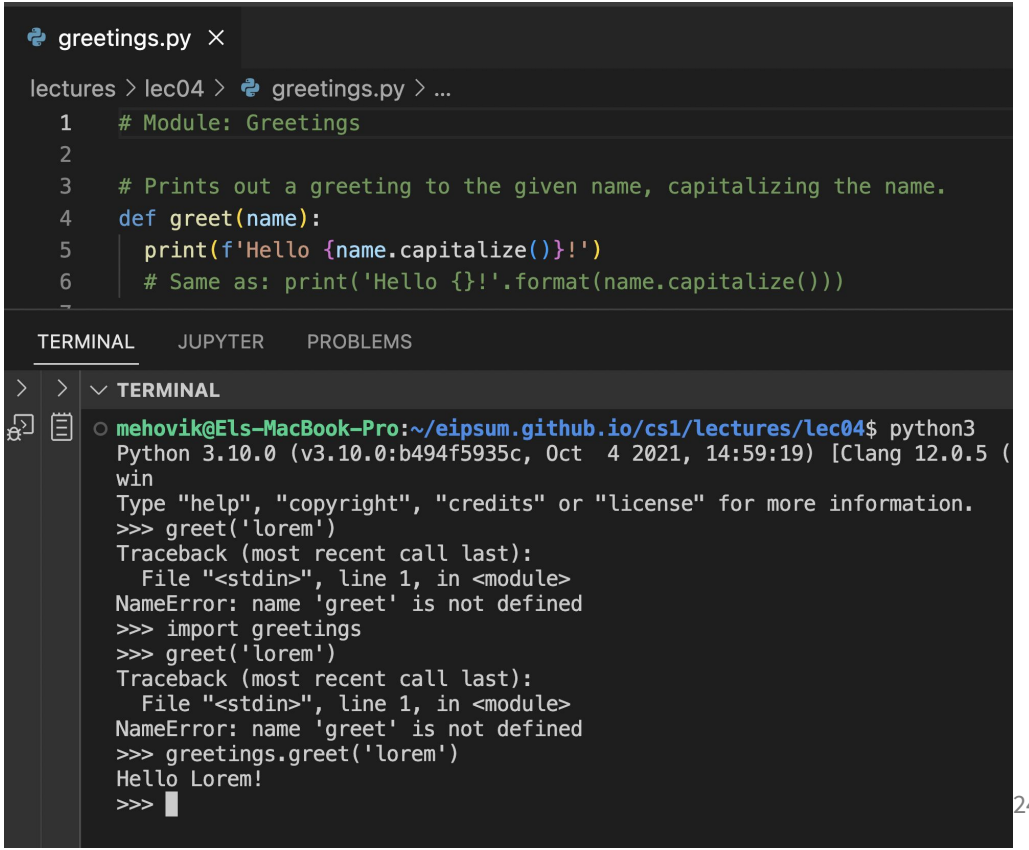
```
# Prints out a greeting to the given name, capitalizing the name.
def greet(name):
  print(f'Hello {name.capitalize()}!')
  # Same as: print('Hello {}!'.format(name.capitalize()))
```

However, Python's **`help()`** function can't use them unless we wrote them in a special way (**docstrings!**)

Aside: Loading Functions in the >>> Interpreter

Remember that running python3 to open a new interpreter *does not* mean that any of your functions written in a file are loaded

To load your functions and test them in the console, you'll need to import them (we'll learn more about **`import`** shortly):



Docstrings

A **docstring** is a regular Python string that is the first thing in any of:

- A function body
- A module
- A class (later in course)

Just like the comments we've seen so far, the docstring doesn't do anything when the program is executed

- But Python stores it as part of the function/module/class

See CS 1 Code Quality Guide for more notes/expectations on docstring format/contents.

Upgrading to Docstrings

```python
# Prints out a greeting to the given name, capitalizing the name.
def greet(name):
  print(f'Hello {name.capitalize()}!')
```

```python
def greet(name):
  """
  Prints out a greeting to the given `name` (str), capitalizing the name.
  """
  print(f'Hello {name.capitalize()}!')
```

Strings are Objects

Python is what's called an **object-oriented** language (we'll learn more about what this means in upcoming lectures)

Most data types are represented as "objects"

An "object" is some **data** with associated **methods** (similar to functions) that work on that data

Python strings are an example of an object

Functions vs. Methods

Functions that are associated with an object are called **methods**

Methods are called on an object using what's called "dot-syntax"

```
>>> 'hello world'.upper()
'HELLO WORLD'
>>>
```

Compare this with the **function `print`**, which takes values (including objects) as arguments:

```
>>> print('hello world')
hello world
>>>
```

Check Your Understanding

Assume the string variable s is defined. Which of the following are function calls?

```
len(s)
s.upper()
s.lower()
help(str)
print(s)
input(s)
"hello {}".format(s)
```

Check Your Understanding

Assume the string variable s is defined. Which of the following are function calls?

```
len(s)
s.upper() # method
s.lower() # method
help(str)
print(s)
input(s)
"hello {}".format(s) # method
```

Practice: String Functions

Problem: Suppose Caltech usernames were automatically generated in the format of first initial followed by full last name (making an unrealistic assumption that everyone has a single first and last name and there are no duplicates). For example, “Lorem Hovik” would generate “lhovik@caltech.edu).

Write a function called **`generate_username`** that helps generate usernames for new Caltech students. Then, add a **`generate_email`** function that uses this function to generate a caltech email address (similar to Lab01's warmup) given a first name and last name.

```
username_to_email('Lhovik')           # returns 'lhovik@caltech.edu'
generate_username('Lorem', 'Hovik')   # returns 'lhovik'
generate_email('Lorem', 'Hovik')      # returns 'lhovik@caltech.edu'
```

More Practice (On Your Own)

The last exercise in HW1 is to write a function called **`gc_content`** which returns the percentage (between 0 and 1) of characters in a given string that are "G" or "C"

Try practicing a related function called **`vowel_count`** which:

- Takes a string as a single argument
- Returns the number of vowels ("a", "e", "i", "o", or "u") in that string
- What if we want to make it case-insensitive (i.e. "A" is treated as "a")?

Upcoming Attractions

Writing our own modules

Lists: Our first data structure!

- Creating lists
- 0-based indexing
- Functions with lists

Loops:

- Processing elements in a sequence (a string or list)
- Repeating code for some number of times with `range`