

01-26-2022

CS051A

INTRO TO COMPUTER SCIENCE WITH TOPICS IN AI

3: Turtle and for loops



David Kauchak

he/him/his

Lectures



Alexandra Papoutsaki

she/her/hers

Lectures



Zilong Ye

he/him/his

Labs

Lecture 3: Turtle and for loops

- ▶ Administrative
- ▶ `print` function
- ▶ Multiline strings and docstrings
- ▶ Turtle module
- ▶ For loops

This week

- ▶ All course handouts can be found on the course website
 - ▶ <https://cs.pomona.edu/classes/cs51a/>
- ▶ [First assignment](#) due this coming Sunday.
- ▶ If you have any questions, join our office hours and mentor sessions.
 - ▶ Schedule is posted on website.
 - ▶ Mentor sessions Zoom links on Slack.

Lecture 3: Turtle and for loops

- ▶ Administrative
- ▶ `print` function
- ▶ Multiline strings and docstrings
- ▶ Turtle module
- ▶ For loops

print function

- ▶ Use it when you want to “print” (i.e. display on the screen) certain expressions (e.g., numbers, strings, contents of variables, messages, etc.).
- ▶ Extremely useful for figuring out how our code works.

```
bbq-functions.py ×  
13  
14 def bbq_cost(angie, jasmine, num_people):  
15     soda_cost = 0.5  
16     hotdog_cost = 0.75  
17  
18     num_hotdogs = hotdogs(angie, jasmine)  
19     num_sodas = soda(num_people)  
20  
21     return num_sodas * soda_cost + num_hotdogs * hotdog_cost
```

Using the `print` function to understand our code

```
>>> bbq_cost(1,2,6)
15.75
```

- ▶ If you wanted to figure out *why* it was that high, you could temporarily add some print statements in the code.

```
bbq-functions.py x
14 def bbq_cost(angie, jasmine, num_people):
15     soda_cost = 0.5
16     hotdog_cost = 0.75
17
18     num_hotdogs = hotdogs(angie, jasmine)
19     num_sodas = soda(num_people)
20
21     print("hotdogs: " + str(num_hotdogs))
22     print("sodas: " + str(num_sodas))
23
24     return num_sodas * soda_cost + num_hotdogs * hotdog_cost
```

```
>>> bbq_cost(1,2,6)
hotdogs: 13
sodas: 12
15.75
```

Don't forget to remove unnecessary print statements

- ▶ We can dig further if we'd like by adding more print statements.
 - ▶ E.g.,

```
print("cost of sodas" + str(num_sodas * soda_cost))
```
- ▶ When you're done, don't forget to *REMOVE ALL PRINT STATEMENTS!*
- ▶ In most cases, we're adding print statements to help us **debug** our program.
 - ▶ **debugging**: the process of finding and removing programming errors.

print vs return

- ▶ print

- ▶ the print function displays the value to the screen/shell.

- ▶ return

- ▶ a return statement has two parts, return [expression]
 - ▶ When the program gets to this line, it evaluates the expression.
 - ▶ Whatever value this expression evaluates to then is "returned" from that function and represents the value at where the function was called.

print_vs_return.py

- ▶ Similar calculations but VERY different behavior.

```
print_vs_return.py x
1  def print_square(number):
2     print(number * number)
3
4
5  def return_square(number):
6     return number * number
```

```
>>> print_square(10)
100
>>> return_square(10)
100
>>> x = print_square(10)
100
>>> x
None
>>> y = return_square(10)
>>> y
100
```

print_vs_return.py

- ▶ `print_square(10)` and `return_square(10)` appear to do the same thing, but they are different.
 - ▶ `print_square(10)` is actually printing to the shell *inside* the function.
 - ▶ `return_square(10)` evaluates to `100`, then that value is printed because the default behavior for the shell is to print the value.
- ▶ This difference is highlighted in the next 4 statements:
 - ▶ `x = print_square(10)` calls `print_square(10)` which prints but does NOT return a value. Therefore, `x` remains undefined.
 - ▶ `y = return_square(10)` calls `return_square(10)` which does NOT print out the value (`100`) but returns it, therefore `y` is assigned the value `100`.

print_vs_return.py

```
# what will happen if the following was included at the bottom
# of the code when we run this program?
print_square(5)
print("#")
return_square(5)
print("##")
print(print_square(5))
print("###")
print(return_square(5))
print("####")
```

▶ If you hit Run (green triangle), you get:

```
25
#
##
25
None
###
25
####
```

print_vs_return.py

- ▶ When you run a file, it starts at the top and executes each statement/line one at a time.
- ▶ `print_square(5)` prints 25.
- ▶ `print("#")` prints #
- ▶ `return_square(5)` does NOTHING. It returns a value, but then we don't do anything with it (just as if we'd typed `5*5` there) so the result of the calculation is lost.
- ▶ `print("##")` prints ##
- ▶ `print(print_square(5))` calls `print_square(5)` which again prints 25. Then, when we return, we try and print out the value that was returned from `print_square(5)`. Since `print_square` does not return a value, we get "None".
- ▶ `print("###")` prints ###
- ▶ `print(return_square(5))` prints 25 because `return_square(5)` returned it!
- ▶ `print("####")` prints ###

return statement

- ▶ When the interpreter reaches a return statement the program indicates a disruption in flow.
- ▶ We have to leave that function.
 - ▶ Therefore any code in a function body that directly follows a return statement cannot be reached.

Lecture 3: Turtle and for loops

- ▶ Administrative
- ▶ `print` function
- ▶ Multiline strings and docstrings
- ▶ Turtle module
- ▶ For loops

Multiline strings

- ▶ So far we've seen double quotes and single quotes to enclose strings.
- ▶ If we want a string to span over multiple lines we have a few options
 - ▶ there is a special character `'\n'` that represents the end of the line. E.g.,

```
multiline_strings.py ×  
1 print("This is a string\nthat spans over multiple\nlines")
```

```
This is a string  
that spans over multiple  
lines
```

Multiline strings using triple quotes

- ▶ Previous approach has a few drawbacks:
 - ▶ hard to read as a human
 - ▶ hard to get formatting/alignment right
 - ▶ if it's a long string (e.g., a paragraph) it's going to go off the screen
 - ▶ pain to copy and paste multiline text from somewhere else
- ▶ Use triple quotes instead, e.g.,

```
multiline_strings.py x
1 print("""This is a multiline string
2     I can continue to type
3     over many different lines
4     and it won't stop until
5     I close the strings""")
```

```
This is a multiline string
I can continue to type
over many different lines
and it won't stop until
I close the strings
```


Docstrings

- ▶ Docstring: a string immediately following a definition.
 - ▶ Another form of commenting.

```
bbq-functions-commented.py x
1  def hotdogs(angie, jasmine):
2      """
3      Returns the number of hotdogs required for the party.
4
5      Parameters:
6      angie -- the number of hotdogs angie will eat
7      jasmine -- the number of hotdogs jasmine will eat
8      """
9      chris = 2 * jasmine
10     brenda = chris - 1
11     wenting = (brenda + 1) // 2 + 1 # add 1 to brenda to round up
12
13     total_hotdogs = angie + jasmine + chris + brenda + wenting
14     return total_hotdogs
```

Using the `help` function to read docstrings

- ▶ If you pass a method as an argument to the `help` function, you will get back the docstring of that method. E.g.,

```
>>> help(hotdogs)
Help on function hotdogs in module __main__:

hotdogs(angie, jasmine)
    Returns the number of hotdogs required for the party.

Parameters:
  angie -- the number of hotdogs angie will eat
  jasmine -- the number of hotdogs jasmine will eat
```

- ▶ This can be VERY useful when you're using code that you haven't written!

Conventions

- ▶ We're going to be defining docstrings for ALL functions we write from here on out.
- ▶ We'll always use triple quotes for docstrings (even if they're just one line).
- ▶ For simple functions, a one line docstring is sufficient.
- ▶ For longer ones, first give a description of what it does, then describe what each of the parameters represents.

Good style

- ▶ Use good variable/function names.
- ▶ Use whitespace (both vertical and horizontal) to make code more readable.
- ▶ Comment code, including both comments and docstrings.
- ▶ Try and write code as simply as possible (more on this as we go).

Lecture 3: Turtle and for loops

- ▶ Administrative
- ▶ `print` function
- ▶ Multiline strings and docstrings
- ▶ Turtle module
- ▶ For loops

Modules

- ▶ **Module**: a collection of functions and variables.
- ▶ Modules allow us to use code that other people have written.
- ▶ For example, there is a module called `math` that has many of the math functions you might want.
- ▶ We can look at the documentation for this module online by searching for "math python" or by going to <https://docs.python.org/3/> and browsing searching there.
 - ▶ <https://docs.python.org/3/library/math.html>
 - ▶ logs
 - ▶ sqrt
 - ▶ trigonometric functions
 - ▶ constants

Importing modules

- ▶ If we want to use a module, we need to tell the program to include it with our program. To do this, we need to "import" it.
- ▶ There are many ways of importing modules (some better than others).
- ▶ For now, we're going to import the functions and variables into our program as if they were local (i.e. just as if we'd written them in our program).
 - ▶ this is convenient for now, but in some situations there are better ways of doing it (more on this later)

```
>>> from math import *
```

- ▶ This statement has multiple components:
- ▶ `from` is a keyword,
- ▶ `math` is the name of the module,
- ▶ `import` loads the module into our program,
- ▶ `*` means everything, i.e. load everything included in the math module.

turtle module

- ▶ The turtle module implements a set of commands similar to the [Logo](#) programming language
- ▶ The basic idea is that you control the movements of a turtle (in our case, it will be an arrow) through basic commands such as:
 - ▶ `forward(distance)`: Move the turtle forward by the specified distance, in the direction the turtle is headed.
 - ▶ `backward(distance)`: Move the turtle backward by distance, opposite to the direction the turtle is headed. Do not change the turtle's heading.
 - ▶ `right(angle)`: Turn turtle right by angle units.
 - ▶ `left(angle)`: Turn turtle left by angle units.
 - ▶ ...and many others
- ▶ As the turtle moves, it draws a line behind it, so by giving it different commands, we can draw things on the screen!
- ▶ Check the [documentation](#) for the turtle class online
- ▶ You'll be getting more comfortable with this documentation as part of next week's lab.

turtle module

- ▶ The turtle module implements a set of commands similar to the [Logo](#) programming language
- ▶ The basic idea is that you control the movements of a turtle (in our case, it will be an arrow) through basic commands such as:
 - ▶ `forward(distance)`: Move the turtle forward by the specified distance, in the direction the turtle is headed.
 - ▶ `backward(distance)`: Move the turtle backward by distance, opposite to the direction the turtle is headed. Do not change the turtle's heading.
 - ▶ `right(angle)`: Turn turtle right by angle units.
 - ▶ `left(angle)`: Turn turtle left by angle units.
 - ▶ ...and many others
- ▶ As the turtle moves, it draws a line behind it, so by giving it different commands, we can draw things on the screen!
- ▶ Check the [documentation](#) for the turtle class online
- ▶ You'll be getting more comfortable with this documentation as part of next week's lab.

Let's move our turtle!

- ▶ How would you create a square?

- ▶ `forward(some_length)`

`right(90)`

`forward(some_length)`

`right(90)`

`forward(some_length)`

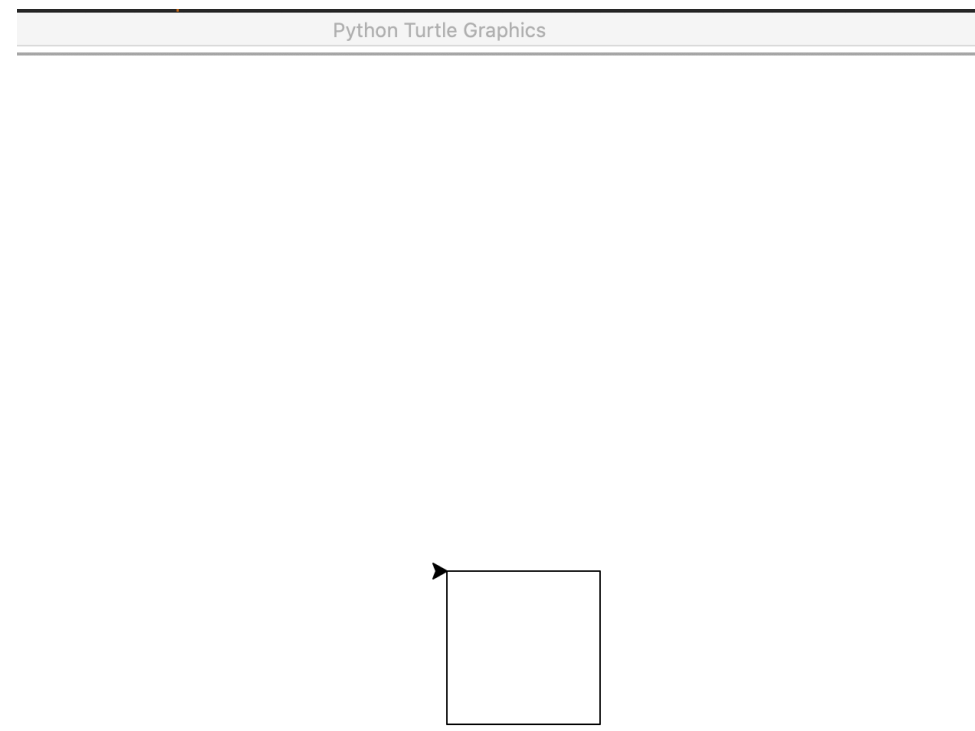
`right(90)`

`forward(some_length)`

Let's move our turtle!

```
turtle-examples.py x
1  from turtle import *
2  from random import randint
3
4
5  def square(length):
6      forward(length)
7      right(90)
8      forward(length)
9      right(90)
10     forward(length)
11     right(90)
12     forward(length)
13     right(90)
```

```
>>> square(100)
```



- ▶ This seems like a lot of repetitive typing. Let's say we can tell the turtle to repeat some statements, would there be a better way of creating a square?
- ▶ go forward some length and then turn right, repeat this 4 times

Lecture 3: Turtle and for loops

- ▶ Administrative
- ▶ `print` function
- ▶ Multiline strings and docstrings
- ▶ Turtle module
- ▶ **For loops**

Python for loops

- ▶ Python has a number of different "loop" structures that allow us to do repetition (computers are really good at doing repetitive tasks!)
- ▶ The for loop is one way of doing this
- ▶ There are a number of ways we can use the for loop, but for now the basic structure we'll use is:

```
for some_variable in range(num_iterations):  
    statement1  
    statement2  
    ...
```

Python for loops syntaxes

```
for some_variable in range(num_iterations):
```

```
    statement1
```

```
    statement2
```

```
    ...
```

- ▶ `for` is a keyword
- ▶ `in` is a keyword
- ▶ `range` is a function that we'll use to tell Python how many repetitions we want
- ▶ `num_iterations` is the number of iterations that we want the loop to do
- ▶ `some_variable` is a local variable whose scope (where it can be referred to) is only within the for loop
 - ▶ `some_variable` will take on the values from `0` to `num_iterations-1` as each iteration of the loop occurs
 - ▶ We're computer scientists so we start counting at zero :)
 - ▶ for example, in the first iteration, it will be `0`, the second time `1`, the third time `2`, etc. we're computer scientists so we start counting at zero :)
- ▶ Don't forget the `:` at the end!
- ▶ Like with defining functions, Python uses indenting to tell which statements belong in the for loop

What would this code do?

```
>>> for i in range(10):  
...     print(i)
```

```
0  
1  
2  
3  
4  
5  
6  
7  
8  
9
```

An iterative square

```
turtle-examples.py ×  
16 def iterative_square(length):  
17     for i in range(4):  
18         forward(length)  
19         right(90)  
20
```


Resources

- ▶ Textbook: Continue reading [Chapter 4](#).
- ▶ [print_vs_return.txt](#)
- ▶ [multiline_strings.txt](#)
- ▶ [bbq-functions-commented.txt](#)
- ▶ [turtle-examples.txt](#)

Practice Problems

- ▶ [Practice 1](#) ([solution](#))

Homework

- ▶ (Work in progress) - Assignment 1