PAYS 2023

INTRODUCTION TO PROGRAMMING USING PYTHON

3: print vs return

Alexandra Papoutsaki

she/her/hers

Lecture 3: print vs return

- `print` function
- Multiline strings and docstrings

`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.

```
def bbq_cost(angie, jasmine, num_people):

    soda_cost = 0.5
    hotdog_cost = 0.75

    num_hotdogs = hotdogs(angie, jasmine)
    num_sodas = soda(num_people)

    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.

```
def bbq_cost(angie, jasmine, num_people):

    soda_cost = 0.5
    hotdog_cost = 0.75

    num_hotdogs = hotdogs(angie, jasmine)
    num_sodas = soda(num_people)

    print("hotdogs: " + str(num_hotdogs))
    print("sodas: " + str(num_sodas))

    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("total cost of hotdogs: " + str(num_hotdogs*hotdog_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.

```
def print_square(number):
    print(number * number)


def return_square(number):
    return number * number
```

```
>>> print_square(10)
100
>>> return_square(10)
100
>>> x = print_square(10)
100
>>> x
>>> 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 4: print vs return

- `print` function
- Multiline strings and docstrings

Multiline strings

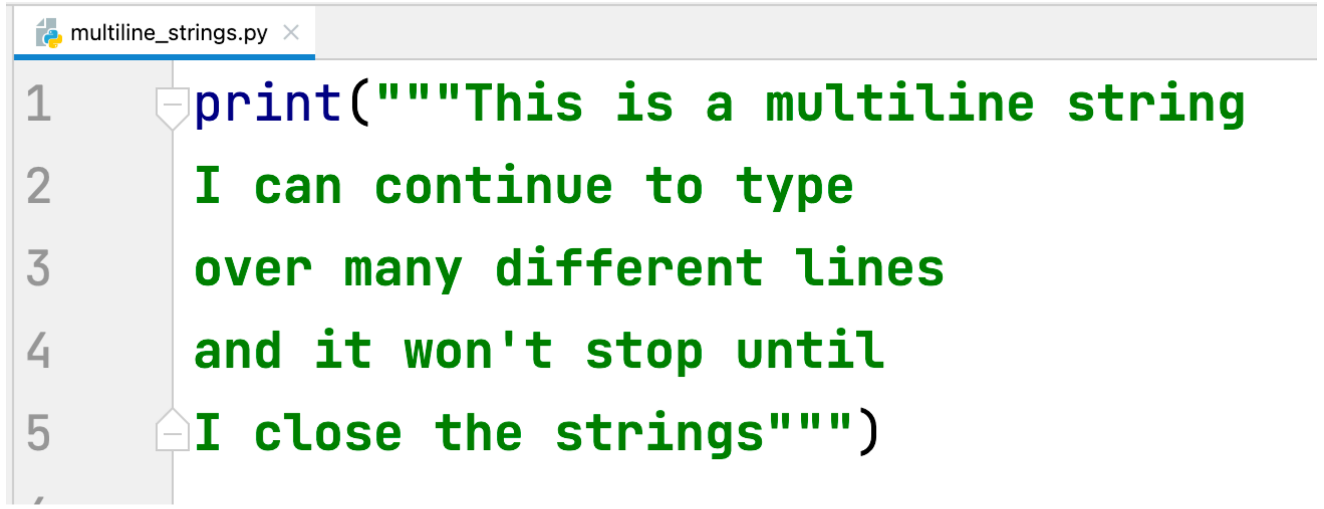
- So far we've seen double quotes and single quotes to enclose strings.
- If we want a string to span over multiple lines we have a few options
  - there is a special character `'\n'` that represents the end of the line. E.g.,

```
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.,

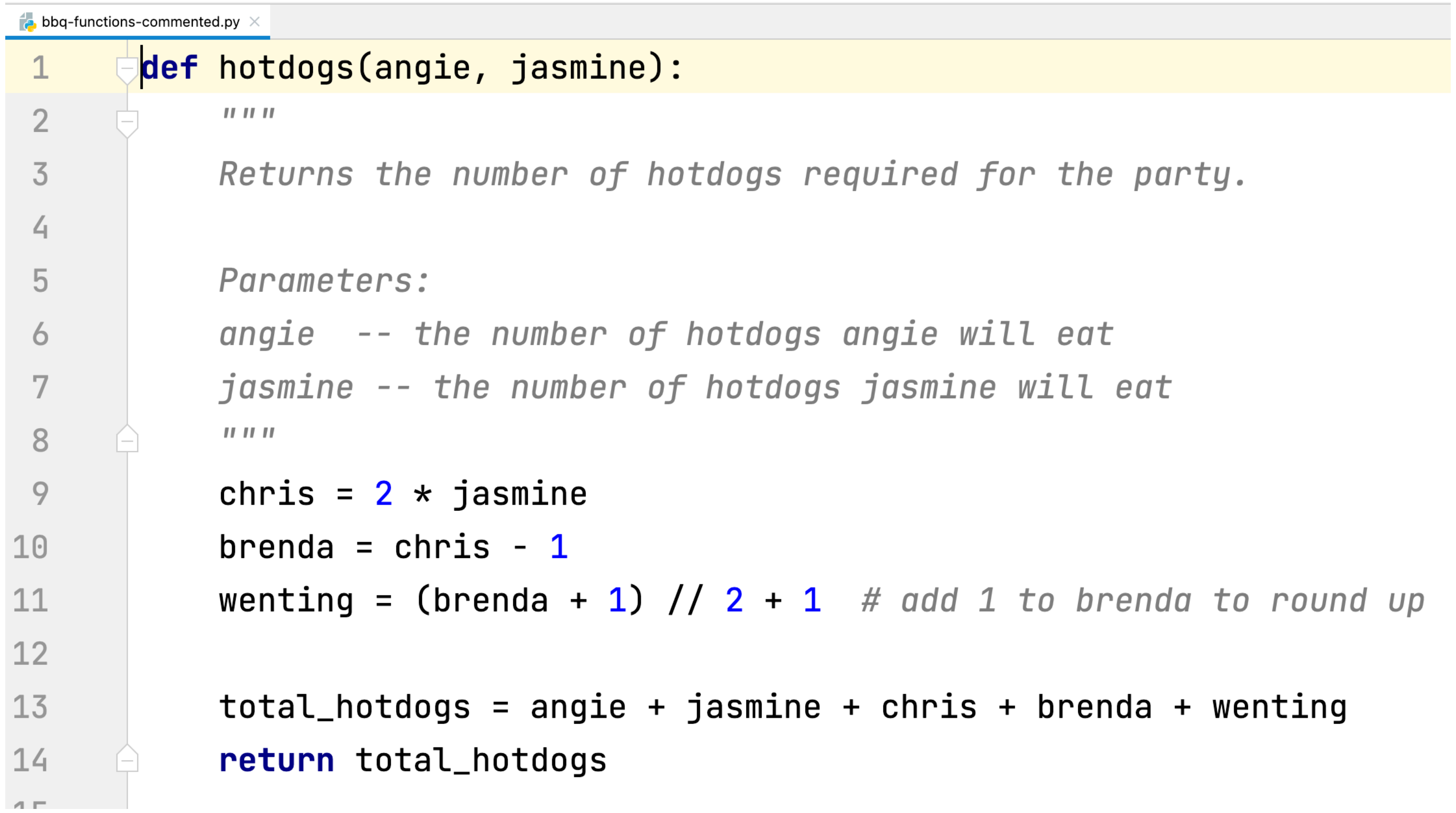


```python
print("""This is a multiline string
I can continue to type
over many different lines
and it won't stop until
I close the strings""")
```

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

Docstrings

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



bbq-functions-commented.py

```python
def 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
    """
    chris = 2 * jasmine
    brenda = chris - 1
    wenting = (brenda + 1) // 2 + 1  # add 1 to brenda to round up

    total_hotdogs = angie + jasmine + chris + brenda + wenting
    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).

Resources

- Textbook: Continue reading Chapter 4.
- print_vs_return.txt
- multiline_strings.txt
- bbq-functions-commented.txt

Practice Problems

- Practice 1 (solution)