CIS 1100

List Comprehensions & Introduction to Functions

Python

Fall 2024

University of Pennsylvania

List Comprehension Syntax

Recall a `for` loop that copies all characters of a string into a list:

```
new_list = []
for character in "ABCD":
    new_list.append(character)
```

"For each character in the string, place that character in the new list I am creating."

👇 👇 👇

```
new_list = [character for character in "ABCD"]
```

List Comprehension Syntax

A basic list comprehension can be written like so:

```
[<expression> for variable in sequence]
```

- `for variable in sequence` works exactly like a regular `for` loop
    - Each element in `sequence` gets visited one-by-one and is given the name `variable`
- The value of `<expression>` is appended to the output list for each element in the sequence
    - Usually write `<expression>` in terms of `variable`
- A new list is created!

Recall: Getting Non-Zero Exam Scores

This loop-based version...

```
exam_scores = [100, 0, 89, 93, 78, 67, 0]
non_zeroes = []                # [] is a list with no contents
for score in exam_scores:      # For each score from the list,
  if score > 0:                # if that score is not zero,
    non_zeroes.append(score)   # add that score to the end of the new list.
```

...can be rewritten to:

```
exam_scores = [100, 0, 89, 93, 78, 67, 0]
non_zeroes = [score for score in exam_scores if score > 0]
print(non_zeroes)
```

🖨️ 👇

```
[100, 89, 93, 78, 67]
```

List Comprehension Practice (L11)

Write the list comprehension so that we have a list containing all of the elements of `values` but increased by 10.

```
values = [0, 5, 10, 23]
values_added_ten = [ FILL IN THIS LIST COMPREHENSION HERE ]
# Should produce a list of [10, 15, 20, 33]
```

Write a list comprehension that makes a list containing all even length strings from `names`:

```
names = ["bob", "steve", "pete", "me", "abcde"]
even_names = [ FILL IN THIS LIST COMPREHENSION HERE ]
# Should produce a list of ["pete", "me"]
```

List Comprehension Practice (L13)

Convert this for loop to a list comprehension that creates an equivalent list in `result`:

```python
strings = ["arriving", "somewhere", "but", "not", "here"]
result = []
for i, string in enumerate(strings):
  new_entry = (" " * i) + string
  result.append(new_entry)
```

```python
strings = ["arriving", "somewhere", "but", "not", "here"]
result = [ FILL IN THIS LIST COMPREHENSION HERE ]
```

CIS 1100

Functions

Python
Fall 2024
University of Pennsylvania

Demystifying Functions

What's happening here?

```
import penndraw as pd
pd.rectangle(0.5, 0.5, 0.1, 0.2)
pd.run()
```

Recall:

- functions are named groups of statements
- those statements are executed when we **call** a function by name

Functions as Named Groups of Statements

```python
def say_hello():
  print("Oh, hello there.")
  print("👋")

print("about to say hello.")
say_hello()
```

🖨️ 👇

```
about to say hello.
Oh, hello there.
👋
```

Activity: Calling Short Functions

Here are two short functions:

```
def middle():
  print(" XXXX ")

def sides():
  print("XX  XX")
```

```
middle()
sides()
middle()
sides()
middle()
```

Draw the shape that gets printed when this program is run. What is it? (S7)

```
middle()
print(" XX")
middle()
sides()
middle()
```

Draw the shape that gets printed when this program is run. What is it? (S8)

Anatomy of a Function

- **Function definitions** consist of the function's signature as well as a block of statements called its **body**
    - A **function signature** consists of:
        - the function's name
        - the list of parameters that it takes as input.

Dissecting a Function

```python
def multiply_two_numbers(a, b):
    print(f"Multiplying {a} x {b}!")
    product = a * b
    return product
```

The **signature**:

```python
def multiply_two_numbers(a, b):
```

- `def`
- the function's name (`multiply_two_numbers`)
- a pair of parentheses
- a comma-separated list of parameters (`a` and `b`)

Dissecting a Function

```python
def multiply_two_numbers(a, b):
    print(f"Multiplying {a} x {b}!")
    product = a * b
    print(product)
```

The **body**:

```python
    print(f"Multiplying {a} x {b}!")
    product = a * b
    print(product)
```

- multiple statements
- all indented one level relative to signature
- uses `a` and `b` as variables without declaring!
- can end with a `return` statement to produce a value (this example doesn't)

Activity: Choosing Function Names

Choose a better name for each of the four functions below. Each function is run with a single list as its input, e.g. `M1([3, 9, 0, 14])`

```python
def M1(lst):
  smallest = lst[0]
  for elem in lst:
    if elem < smallest:
      smallest = elem
  print(smallest)

def M2(lst):
  running_sum = 0
  for elem in lst:
    running_sum += 1
  print(running_sum)
```

```python
def M3(lst):
  saved = lst[0]
  for elem in lst:
    if elem > saved:
      saved = elem
  print(saved)

def M4(lst):
  running_sum = 0
  for elem in lst:
    running_sum += elem
  print(running_sum)
```

A: `max`, B: `min`, C: `sum`, D: `len`

Recap: Calling Functions with Inputs

Here is a function that takes a message and a number and prints that message that number of times.

```python
def print_n_times(msg, n):
    counter = 0
    while counter < n:
        print(msg)
        counter = counter + 1
```

What happens when we call the function: `print_n_times("Hi!", 3)`?

Recap: Calling Functions with Inputs

- The function's *parameters* are `msg` and `n`.
  - These are names for variables that can be used in the body of the function
- The function call provides two **arguments**: `"Hi!"` and `3`
  - These are the values that the parameter variables will take at the start of the function execution.

```
# calling print_n_times("Hi!", 3)
def print_n_times(msg, n):
    # msg = "Hi!"
    # n = 3
    counter = 0
    while counter < n: # while counter < 3:
        print(msg)      # print("Hi!")
        counter = counter + 1
```

Activity: Counting Numbers

```python
def add_three_numbers(a, b, c):
  first_two = a + b
  last = c + first_two
  print(last)
```

- M5: calling the function as `add_three_numbers(3, 4, 7, 9)` leads the program to immediately crash
- M6: calling the function as `add_three_numbers("three", "four", "five")` leads the program to immediately crash

A: True, B: False

Activity: Working Towards Writing a Function

Assuming you have a list `lst` containing a bunch of numbers, write a couple of loops that print out all of the **negative** numbers and then all of the **non-negative numbers**. (C14, but leave just a little space at the top)

e.g.

```
lst = [9, -19, 31, -13, 1, 2]
# TODO: Your loop(s) here
```

🖨️ 👇

```
-19 -13 9 31 1 2
```

You're not writing a whole function yet! Just write some lines & loops like you've been doing before.

Activity: Working Towards Writing a Function

Write the signature for a function that prints out all of the **negative** numbers and then all of the **non-negative numbers**. (L15)

Remember: a signature consists of a `def`*, a function name, and a list of parameters the function should be called with.*

Activity: Working Towards Writing a Function

Add a signature to the code you wrote for (C14) in order to turn it into a function that can be called.

Then, in (C16), write an example of a function call that would print out the following output:

```
-30 -14 3 19 8
```

New: `return`

Function calls are themselves *expressions*, meaning that they always have a value.

- The value of a function call is determined by the value that function **returns**

`return` is keyword that serves two purposes:

- stops function execution in its tracks
- provides a value for the expression of the function call

`return`: An Example

```python
def multiply_two_numbers(a, b):
    print(f"Multiplying {a} x {b}!")
    product = a * b
    return product
```

If we write the call `multiply_two_numbers(3, 7)`, then...

```python
    # a = 3
    # b = 7
    print(f"Multiplying {a} x {b}!")
    product = a * b                     # product = 3 * 7
    return product                      # return 21
```

...we return the value of `product`, which is `21` based on

this function call. The following therefore evaluates to `True`:

```python
multiply_two_numbers(3, 7) == 21
```

Printing vs. Returning

An output that's *printed* is not the same as an output that's *returned.*

- Any call to `print()` will make text appear on the screen, but it doesn't produce a value
- If a function is supposed to calculate and create some value (e.g. the product of two numbers), it must *return* that value in the function body.

Functions that Have No `return`

```
def our_min(lst):
  smallest = lst[0]
  for elem in lst:
    if elem < smallest:
      smallest = elem
  print(smallest)

def our_len(lst):
  running_sum = 0
  for elem in lst:
    running_sum += 1
  print(running_sum)
```

```
some_numbers = [1000, 3, 8]

result = our_min(some_numbers) # 🖨️➡️ 3
print(result)                # 🖨️➡️ None

result = our_len(some_numbers) # 🖨️➡️ 3
print(result)                # 🖨️➡️ None
```

These functions both *compute* some value and then *print* it but do not *return* it.

Adding `return`

```python
def our_min(lst):
  smallest = lst[0]
  for elem in lst:
    if elem < smallest:
      smallest = elem
  return smallest)

def our_len(lst):
  running_sum = 0
  for elem in lst:
    running_sum += 1
  return running_sum)
```

```python
some_numbers = [1000, 3, 8]

result = our_min(some_numbers) # 🖨➡ Nothing!
print(result)                  # 🖨➡ 3

result = our_len(some_numbers) # 🖨➡ Nothing!
print(result)                  # 🖨➡ 3
```

These functions now *compute* some value and then *return* it but do not *print* it.

The Point of No `return`?

`return` works as a stopping/exit point for your program. If you execute a line with `return`, you will leave that function call execution.

```
def print_all_above(lst, k):
  for elem in lst:
    if elem > k:
      print(elem)

print_all_above([5, 10, 15], 8)
```

🖨️ 👇

```
10 15
```

The Point of No `return`?

`return` works as a stopping/exit point for your program. If you execute a line with `return`, you will leave that function call execution.

```python
def print_first_above(lst, k):
  for elem in lst:
    if elem > k:
      print(elem)
      return

print_all_above([5, 10, 15], 8)
```

🖨️ 👇

```
10
```

The Point of No `return`?

`return` works as a stopping/exit point for your program. If you execute a line with `return`, you will leave that function call execution.

```python
def return_first_above(lst, k):
  for elem in lst:
    if elem > k:
      return elem

print_all_above([5, 10, 15], 8)
```

🖨️ 👇

```
```

...but it does return `10`!