Why Does My Computer Do That? Intro to Coding with Python– Mathematical Operators

Dr. Ab Mosca (they/them)

Slides based off slides courtesy of Jordan Crouser (https://jcrouser.github.io/)

Plan for Today

- More mathematical operators
- Formatting print statements

(RECAP) Core concept 2: numeric values

- Two kinds of **numbers** in CS:
  - integers ("whole numbers")
  - floats ("decimals" or "floating point numbers")
- Basic **operators**:
  - addition: `+`
  - subtraction: `-`
  - multiplication: `*`
  - division: `/`
  - integer division: `//`
  - exponentiation: `**` (power)
  - modular arithmetic: `%` (modulo)

(RECAP) Core concept 2: numeric values

- Two kinds of **numbers** in CS:
  - integers ("whole numbers")
  - floats ("decimals" or "floating point numbers")
- Basic **operators**:
  - addition: `+`
  - subtraction: `-`
  - multiplication: `*`
  - division: `/`
  - integer division: `//`
  - exponentiation: `**` (power)
  - modular arithmetic: `%` (modulo)

Reviewing integer operators: // and %

What is the result of the following operations?

```
21 // 5
21 % 5
9 // 3
9 % 3
13 // 5
13 % 5
139 // 20
139 % 20
```

Reviewing integer operators: // and %

What is the result of the following operations?

```
21 // 5     # 4
21 % 5      # 1
9 // 3      # 3
9 % 3       # 0
13 // 5     # 2
13 % 5      # 3
139 // 20   # 6
139 % 20    # 19
```

Built-in functions that work on numbers

- `abs(x)` # return the absolute value of x
- `float(x)` # return x parsed as a float
- `int(x)` # return x parsed as an int
- `max(…)` # return the largest of a list of numbers
- `min(…)` # return the smallest of a list of numbers
- `round(x[, n])` # return *x* rounded to *n* digits after the
  # decimal point. If *n* is omitted, it
  # returns the nearest integer value
- `sum(…)` # return the sum of a list of numbers

Aside: what does **parsed** mean?

- `abs(x)` # return the absolute value of x
- `float(x)` # return x **parsed** as a float
- `int(x)` # return x **parsed** as an int
- `max(…)` # return the largest of a list of numbers
- `min(…)` # return the smallest of a list of numbers
- `round(x[, n])` # return *x* rounded to *n* digits after the
  # decimal point. If *n* is omitted, it
  # returns the nearest integer value
- `sum(…)` # return the sum of a list of numbers

Aside: what does `return` mean?

- `abs(x)` # **return** the absolute value of x
- `float(x)` # **return** x parsed as a float
- `int(x)` # **return** x parsed as an int
- `max(…)` # **return** the largest of a list of numbers
- `min(…)` # **return** the smallest of a list of numbers
- `round(x[, n])` # **return** *x* rounded to *n* digits after the
  # decimal point. If *n* is omitted, it
  # returns the nearest integer value
- `sum(…)` # **return** the sum of a list of numbers

RECAP: Keywords

- Some words in Python* are reserved as keywords, and cannot be used as a variable name:

```
and as assert break class continue def del elif else
 except exec finally for from global if import in is
lambda not or pass raise return try while with yield
```

a reserved keyword

* other languages have their own set of reserved words

Peek ahead: "functions"

demo6.py - /Users/jcrouser/Google Drive/Teaching/Course Material/SCS-Noonan...

```
def do_something():
    # perform some operations, like:
    x = 2 + 3

    # send stuff back to the main program
    return x

y = do_something()
print(y) # y = 5
```

Ln: 9 Col: 16

The `math` module

- Lots of other things we might want to do with numerical values are available as functions in the **`math`** module

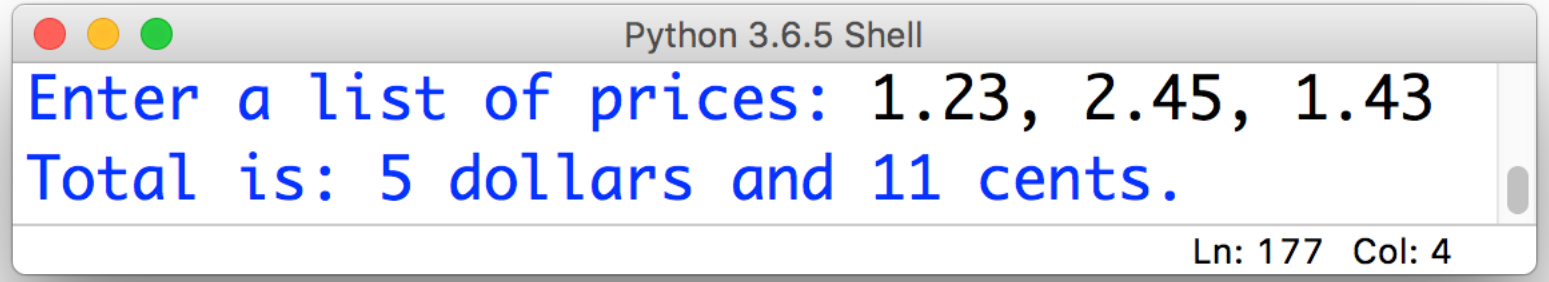
  - In Python, modules are just files containing Python definitions and statements (ex. ***`name`***`.`**`py`**)
  - These can be imported using **`import`** ***`name`***
  - To access **`name`**'s functions, type ***`name.function()`***

- `import math`
  - `math.floor(f)` # round float f down
  - `math.ceil(f)` # round float f up
  - `math.sqrt(x)` # take the square root of x

And more! Check out:
https://docs.python.org/2/library/math.html

15-minute exercise: dollars and cents

Use **built-in functions** and functions from the **`math` module** to take a list of prices, calculate their sum, and output their total formatted like this:



Finishing touches…

- What we have now:

```
Python 3.6.5 Shell
Enter a list of prices: 1.23, 2.45, 1.43
Total is: 5 dollars and 11 cents.
Ln: 177  Col: 4
```

- What we probably want:

```
Python 3.6.5 Shell
Enter a list of prices: 1.23, 2.45, 1.43
Total is: $5.11
Ln: 181  Col: 4
```

Finishing touches…

- What we have now:

```
Python 3.6.5 Shell
Enter a list of prices: 1.23, 2.45, 1.43
Total is: 5 dollars and 11 cents.
Ln: 177  Col: 4
```

- What we probably want:

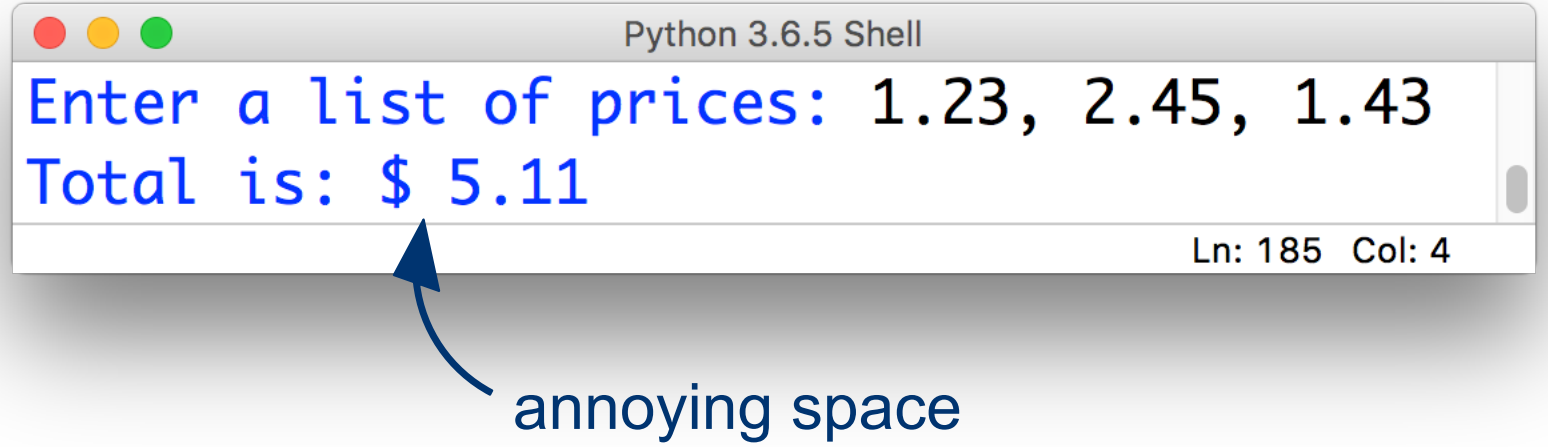
```
Python 3.6.5 Shell
Enter a list of prices: 1.23, 2.45, 1.43
Total is: $5.11
Ln: 181  Col: 4
```

Ideas? What tools do you have to achieve this?

Close, but not quite:

- Just using concatenation:

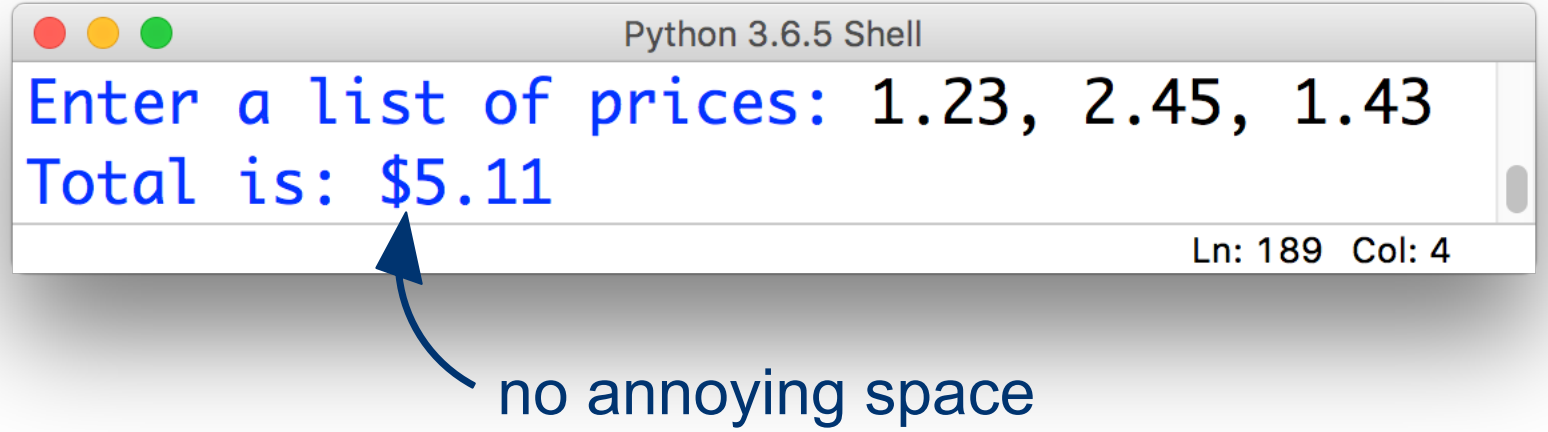
```
print("Total is: $", sum(x)))
```



Closer, but unsatisfying (and fragile)

- Using concatenation and casting to string:

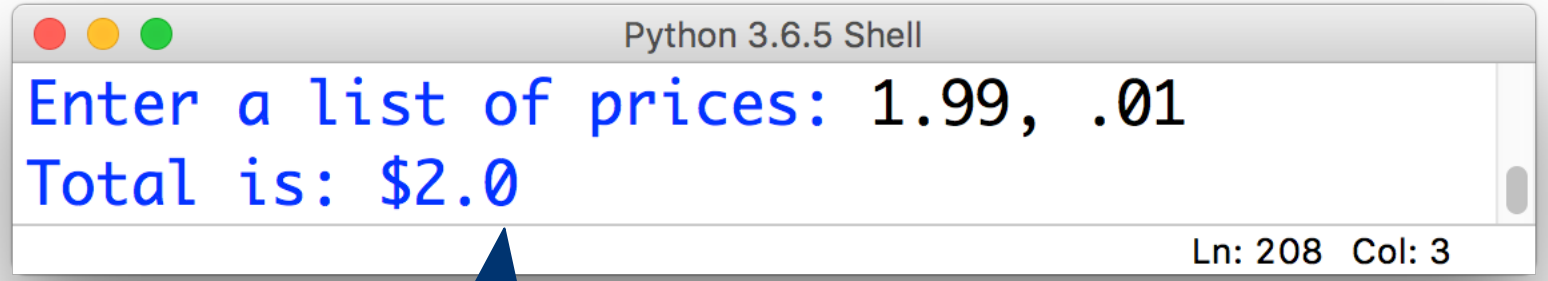
```
print("Total is: $",str(sum(x))))
```



Closer, but unsatisfying (and fragile)

- Using concatenation and casting to string:

```
print("Total is: $", str(sum(x))))
```



wrong number of decimal places

Solution: formatting with `.format()`

- The `.format()` method (which gets called on a `string`) might be helpful here!

Solution: formatting with `.format()`

- The **`.format()`** method (which gets called on a **`string`**) might be helpful here!
- How it works:

insert

*demo6.py - /Users/jcrouser/Google Drive/Teaching/Course Material/SCS-...

```
name = "Jordan"
print("| Name: {0:10} |".format(name))
```

Ln: 1 Col: 16

print (at least) 10 characters

Result: `| Name: Jordan |`

Solution: formatting with `.format()`

- The **`.format()`** method (which gets called on a **`string`**) might be helpful here!
- How it works:

*demo6.py - /Users/jcrouser/Google Drive/Teaching/Course Material/SCS-...

```
name = "Al"
print("| Name: {0:10} |".format(name))
```

Ln: 1 Col: 10

Result: `| Name: Al  |`

Solution: formatting with `.format()`

- The **`.format()`** method (which gets called on a **`string`**) might be helpful here!
- How it works:

demo6.py - /Users/jcrouser/Google Drive/Teaching/Course Material/SCS-...

```
name = "Theresa-Marie"
print("| Name: {0:10} |".format(name))
```

Ln: 1 Col: 21

Result: `| Name: Theresa-Marie |`

doesn't truncate

Calling `.format()` with multiple inputs

- Can also handle multiple inputs, e.g.

demo6.py - /Users/jcrouser/Google Drive/Teaching/Course Material/SCS-Noonan-CSC/labs/demo6.py (3.6.5)

```
first = "Jordan"
last = "Crouser"
print("| Name: {0:10} {1:10} |".format(first, last))
```

Ln: 3 Col: 24

0th 1st

put the 1st thing here

put the 0th thing here

Result: `| Name: Jordan Crouser |`

Right-justification with `.format()`

- To align the format to the right instead of to the left, use **>**

demo6.py - /Users/jcrouser/Google Drive/Teaching/Course Material/SCS-Noonan-CSC/labs/demo6.py (3.6.5)

```
first = "Jordan"
last = "Crouser"
print("| Name: {0:>10} {1:10} |".format(first, last))
```

Ln: 3 Col: 19

Result: `| Name:  Jordan Crouser  |`

`.format()` on integers

- Calling `.format()` on an integer works just like with strings, but they're automatically right-aligned

demo6.py - /Users/jcrouser/Google Drive/Teaching/Course Material/...

```
age = 32
print("| Age: {0:3} |".format(age))
```

Ln: 1 Col: 3

Result: `| Age:  32 |`

`.format()` on integers

- Use < to left-align:

demo6.py - /Users/jcrouser/Google Drive/Teaching/Course Material/...

```
age = 32
print("| Age: {0:<3} |".format(age))
```

Ln: 2 Col: 18

Result: `| Age: 32  |`

`.format()` on floats

- We need to specify a number of digits **before** and **after** the decimal point:

demo6.py - /Users/jcrouser/Google Drive/Teaching/Course Material/SCS-No...

```
num = 327.211932
print("| Age: {0:3.2f} |".format(num))
```

Ln: 2 Col: 21

Result: `| Age:  327.21 |`

Revisiting: dollars and cents

Modify your previous code to use the `.format()` method so that your output looks like this:

```
Python 3.6.5 Shell
Enter a list of prices: 1.23, 2.45, 1.43
Total is: $5.11
Ln: 189  Col: 4
```