Intro to Computer Science

Andey Robins & Mike Borowczak

Reinforcement Week 1

Outline

- Temperatures
- Don’t Repeat Yourself (DRY)
- More About Types
- More About Functions

Previously

```
pb_per_sandwich = 4
j_per_sandwich = pb_per_sandwich - 1
print(pb_per_sandwich, "oz of PB",
        j_per_sandwich, "oz of jelly")

pb_per_sandwich = 8
print(pb_per_sandwich, "oz of PB",
        j_per_sandwich, "oz of jelly")
```

Idea: We can combine variables with values to calculate new values.

Temperature Calculator

Celsius to Fahrenheit

$$Y = (X \times \frac{9}{5}) + 32$$

And this equation has 2 variables:

```
given_temp_in_c    # X
calc_temp_in_f     # Y
```

```
given_temp_in_c = 0
# we want calc_temp_in_f to be 32
calc_temp_in_f = 32
```

Convert an Equation to Python

$$(X \times \frac{9}{5}) + 32$$

Which in code is:

```
given_temp_in_c * 9 / 5 + 32
```

Clarify Order of Operations

```
(given_temp_in_c * (9 / 5)) + 32
```

Assign to a Variable

```
calc_temp_in_f = (given_temp_in_c
                  * (9 / 5)) + 32
```

Put it together in a program

```
given_temp_in_c = 100
calc_temp_in_f  = given_temp_in_c * 9/5 + 32
print(given_temp_in_c, "C is ", calc_temp_in_f,"F")
```

Test Values

C	F
0	32
100	212
20	68
-20	-4
-100	-148

Questions

What's the Problem?

This is kinda difficult to calculate all these values. Running all three lines of code each time, by hand, is a pain, and wouldn't it be nice if there was a better way?

The Better Way

Don’t Repeat Yourself

AKA D.R.Y.

DRY

"'Don't repeat yourself' is a principle of software development aimed at reducing repetition of software patterns, replacing it with abstractions or using data normalization to avoid redundancy." - Wikipedia

DRY

"'Don't repeat yourself' is a principle of software development aimed at ***reducing repetition*** *of software patterns, replacing it with abstractions or using data normalization* ***to avoid redundancy.****"* - Wikipedia

A Better Definition

"The DRY principle is a best practice in software development that recommends software engineers to do something once, and only once." - Laura Fitzgibbons via WhatIs.com

A Better Definition

"The DRY principle is a ***best practice*** *in software development that recommends software engineers* ***to do something once, and only once.****"* - Laura Fitzgibbons via WhatIs.com

Functions

You've already seen us use a few functions!!

```
type(123456)
type("This is a string")

print("Hello world!")
print("My name is Andey")
```

Or even this one:

```
codio@stormeternal-cornereverest:~/workspace$ python3
Python 3.6.9 (default, Apr 18 2020, 01:56:04)
[GCC 8.4.0] on linux
Type "help", "copyright", "credits" or "license" for more i
>>> quit()
codio@stormeternal-cornereverest:~/workspace$
```

- `type(...)` is the function to get the type of a value
- `print(...)` is the function to display a value in the terminal
- `quit()` is a function in the interpreter to leave the interpreter

Anatomy of a Function Call

How do we invoke (or call) a function?

- Write the function's name
- An open parenthesis '('
- Any arguments the function may take
- A closing parenthesis ')'

i.e. `type(1.414)`

What is an Argument?

An **argument** is a value that is given to a function.

```
# the argument is the string "Hello World"
print("Hello world")
# the argument is the float 1.732
type(1.732)
# this function takes no arguments
quit()
```

What do we want?

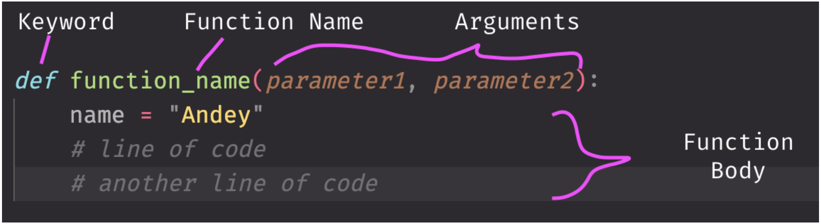
We want to be able to call a function with the celsius value and return the fahrenheit conversion.

```
# something like this
given_temp_in_c = 100
calc_temp_in_f  = convert_c_to_f(given_temp_in_c)
print(given_temp_in_c, "C is ", calc_temp_in_f,"F")
```

The Problem

The problem with the code on the previous slide: python doesn't have a function called `convert_c_to_f`.

The good news, is we can create a function!



```
def function_name(parameter1, parameter2):
    name = "Andey"
    # line of code
    # another line of code
```

Figure 1: The anatomy of a function

1. The function begins with the keyword `def`
2. The name of the function comes next
3. Parentheses are placed
4. Provide the names of any arguments
5. End the line with a colon
6. Indent the “body” of the function

convert_c_to_f

```python
def convert_c_to_f(temp_in_c):
    temp_in_f = temp_in_c * (9 / 5) + 32
```

What happens when we run our code?

```
# something like this
given_temp_in_c = 100
calc_temp_in_f  = convert_c_to_f(given_temp_in_c)
print(given_temp_in_c, "C is ", calc_temp_in_f,"F")
```

```
100 C is  None F
```

Why None?

After we call our function, the value of `temp_in_f` doesn't go anywhere! It only exists within our function.

This is something called *scope* and we will cover it in more detail in the future. For now, we just need to tell our code to return the value of `temp_in_f` to where we called our function.

```python
def convert_c_to_f(temp_in_c):
    temp_in_f = temp_in_c * (9 / 5) + 32
    return temp_in_f
```

We can even apply the idea of DRY to refactor out the `temp_in_f` variable.

```python
def convert_c_to_f(temp_in_c):
    return temp_in_c * (9 / 5) + 32
```

Now let's rewrite some of our earlier work using our new function!

```
givenTempinC = 0
convertedTempinF  = convert_c_to_f(givenTempinC)
print(givenTempinC, "C is ", convertedTempinF,"F")

givenTempinC = 100
convertedTempinF  = convert_c_to_f(givenTempinC)
print(givenTempinC, "C is ", convertedTempinF,"F")

givenTempinC = 20
convertedTempinF  = convert_c_to_f(givenTempinC)
print(givenTempinC, "C is ", convertedTempinF,"F")
```

And even apply DRY principles. . .

```
givenTempinC = 0
print(givenTempinC, "C is ",
        convert_c_to_f(givenTempinC),"F")

givenTempinC = 100
print(givenTempinC, "C is ",
        convert_c_to_f(givenTempinC),"F")

givenTempinC = 20
print(givenTempinC, "C is ",
        convert_c_to_f(givenTempinC),"F")
```

Questions?

More About Types

More about Types

```
>>> type(1.23)
<class 'float'>
>>> type('hi!')
<class 'str'>
>>> type(1 + 2)
<class 'int'>
```

How can we combine different types?

All of these examples use the ‘+’ (plus) operator

- What does a ‘string’ + ‘string’ give us?
- How about ‘int’ + ‘float’?
- ‘string’ + ‘int’?
- ‘int’ + ‘string’?

- ‘string’
- ‘float’
- TypeError: can only concatenate str (not “int”) to str
- TypeError: unsupported operand type(s) for $+$: ‘int’ and ‘str’

Can you subtract types?

- 1.3 - 2
- 1.1 - 1
- 2 - 1
- ‘asdf’ - ‘f’
- ‘asdf’ - 17

- ‘float’
- ‘float’
- ‘int’
- ???
- ???

- ‘float’
- ‘float’
- ‘int’
- TypeError: unsupported operand type(s) for -: ‘str’ and ‘str’
- TypeError: unsupported operand type(s) for -: ‘str’ and ‘str’

Can you multiply types?

Let's assume that any combination of numbers continues to work (i.e. float & int, int & int, etc.)

- `'a' * 'a'`
  - `<str> * <str>`
- `'a' * 3`
  - `<str> * <int>`

- TypeError: can’t multiply sequence by non-int of type ‘str’
- ‘aaa’
  - `<str>`

How do you know what type an action has?

```
>>> type('a' * 3)
<class 'str'>
```

Think Types are Cool?

I do! And there's lots of other people who think so too! (like at least 7 of us)

There's a whole field of study about types and how they relate to programming languages called *Type Theory*, and it's something I use in my research. Don't worry, there's plenty of interesting things to continue learning about types as we go along in the course!

Questions?

More about Functions

More about Functions

Coming soon to a classroom near you!