Functions and Recursion

Camp Inc. Coding Track

Summer 2016

Warm Up: Spot the Bug(s)!

What is wrong with the snippet of code below?
The code is intended to double all the numbers in a list `my_list`.

```
# Assume my_list is a list containing only ints
i = 0
while i <= len(my_list):
    my_list[i] = my_list[i] * 2
    i = i + 1
```

Warm Up: Spot the Bug(s)!

Which alternative would you choose?

```
1 i = 0
  while my_list < len(my_list):
      my_list[i] = my_list[i] * 2
      i = i + 1
```

```
2 for item in my_list:
      item = item * 2
```

```
3 for i in range(len(my_list)):
      my_list[i] = my_list[i] * 2
```

```
4 my_list = [x * 2 for x in my_list]
```

Warm Up: Spot the Bug(s)!

Which alternative would you choose?

1
```
i = 0
while my_list < len(my_list):
    my_list[i] = my_list[i] * 2
    i = i + 1
```

2
```
for item in my_list:
    item = item * 2
```

Bug!

3
```
for i in range(len(my_list)):
    my_list[i] = my_list[i] * 2
```

4
```
my_list = [x * 2 for x in my_list]
```

Some Functions you Already Know

We've already been using functions for a while now. Here are some you already know:

- `print(value, ...)` – Writes all of its arguments to the console on a single line separated by spaces

Some Functions you Already Know

We've already been using functions for a while now. Here are some you already know:

- `print(value, ...)` – Writes all of its arguments to the console on a single line separated by spaces
- `input(prompt)` – Prompts a user for input and returns the string they typed

Some Functions you Already Know

We've already been using functions for a while now. Here are some you already know:

- `print(value, ...)` – Writes all of its arguments to the console on a single line separated by spaces
- `input(prompt)` – Prompts a user for input and returns the string they typed
- `len(sequence)` – Returns the length of the provided sequence

Making Our Own Functions

Python allows us to make our own functions. Here is a simple example:

```
def greet(name):
    print("Nice to meet you,", name)
```

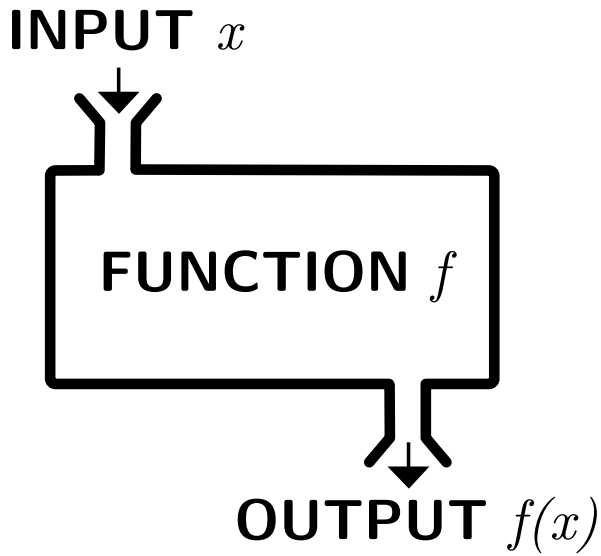
We can then call the function like any other function:

```
greet("Alice")
greet("Bill")
```

```
Nice to meet you, Alice
Nice to meet you, Bill
```

Why Make Functions?

- With functions, we can clean up repetitive code by combining common features
- With functions, we can hide the implementation to the programmer to simplify programming



Key Point

Functions provide the power of **abstraction**. This allows us to preform similar operations without needing to use separately coded parts.

Indentation Still Determines Scope

```
def my_fun():
    print("Inside my_fun!")
    print("When does this print?")
my_fun()
```

```
def my_fun():
    print("Inside my_fun!")
print("When does this print?")
my_fun()
```

Indentation Still Determines Scope

```
def my_fun():
    print("Inside my_fun!")
    print("When does this print?")
my_fun()
```

```
Inside my_fun!
When does this print?
```

```
def my_fun():
    print("Inside my_fun!")
print("When does this print?")
my_fun()
```

```
When does this print?
Inside my_fun!
```

Variables in a Function

Variables inside a function are accessible only to that function call. Attempts to access those variables outside the function will result in an exception.

```
def hello(name):
    sentence = "Hello, " + name + "!"
    print(sentence)

hello("John")
print(sentence)      # this will cause an error
```

Returning from a Function

Just like `input` returns a string, your functions may return a value as well. Consider the following simple example:

```
def add_em(x, y):
    result = x + y
    return result
```

Calling `print(add_em(12,15))` would print:

```
27
```

Returning Immediately Ends a Function Call

As soon as a **return** statement is encountered, the function call will immediately return and will not continue to execute.

This means the following example will **not** print anything at all.

```
def add_and_print(x, y):
    result = x + y
    return result
    print("The sum is", result)

add_and_print(12, 15)
```

Recursion: What is it?

Recursive functions are functions which rely on themselves to calculate part of the answer. Recursive functions usually have a base case that causes the recursion to end. Here is an example as a story:

A child couldn't sleep, so her mother told a story about a little frog,
who couldn't sleep, so the frog's mother told a story about a little bear,
who couldn't sleep, so bear's mother told a story about a little weasel
...who fell asleep.
...and the little bear fell asleep;
...and the little frog fell asleep;
...and the child fell asleep.

Recursive Functions in Python

Consider the factorial operation.

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 1$$

Recursive Functions in Python

Consider the factorial operation.

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 1$$

We could define this recursively as:

- **Base case:** $1! = 1$
- **Recursive part:** $n! = n(n-1)!$

Recursive Functions in Python

Consider the factorial operation.

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 1$$

We could define this recursively as:

- **Base case:** $1! = 1$
- **Recursive part:** $n! = n(n-1)!$

To code this as a recursive function in Python, we could do:

```
def fact(n):
    if n == 1:
        return 1
    return n*fact(n-1)
```

Practice: Doubling Function

Practice by writing a function which takes a list as its parameter, doubles all elements in the list, and returns the doubled list.

```
def double_elements(my_list):
    # your code here

print(double_elements([1, 2, 3, 4]))
print(double_elements([7, 14, 21, 28, 35]))
print(double_elements([42, 42, 42]))
print(double_elements([]))
```