

Functions

- Functions are groups of statements to which you give a name.
 - "Defining" a function; uses the "def" keyword.
- That group of statements can then be referred to by that name later in the program.
 - "Calling" a function; uses the name of the function then an opening/closing set of parentheses.

Functions: an example

```
# These 3 lines define the rap function.
```

```
def rap():  
    print("Now this is the story all about how")  
    print("My life got flipped turned upside-down")
```

```
# These 4 lines define the main function.
```

```
def main():  
    rap()           # This line calls rap().  
    print()  
    rap()           # This line calls rap() a 2nd time.
```

```
main()           # This line calls main() to start the program.
```

A different example

- You want to write a program to sing the "Happy Birthday" song to the user.
- Here's a first attempt at doing this using functions...

```
# THIS PROGRAM DOESN'T WORK!
```

```
def sing_song():  
    print("Happy birthday to you! \  
           Happy birthday to you!")  
    print("Happy birthday dear", name, \  
          "Happy birthday to you!")  
  
def main():  
    name = input("What is your name? ")  
    sing_song()  
  
main()
```

Local variables

- Every variable assigned to inside a function is "owned" by that function.
- It is invisible to all other functions in your program except its owner.
- These are called *local variables* because they can only be used "locally" (within their own function).

```
# THIS PROGRAM DOESN'T WORK!
```

```
def sing_song():  
    print("Happy birthday to you! \  
        Happy birthday to you!")  
    print("Happy birthday dear", name, \  
        "Happy birthday to you!")
```

```
def main():  
    name = input("What is your name? ")  
    sing_song()
```

```
main()
```

name is a local variable – it is invisible to Python outside of the main function.

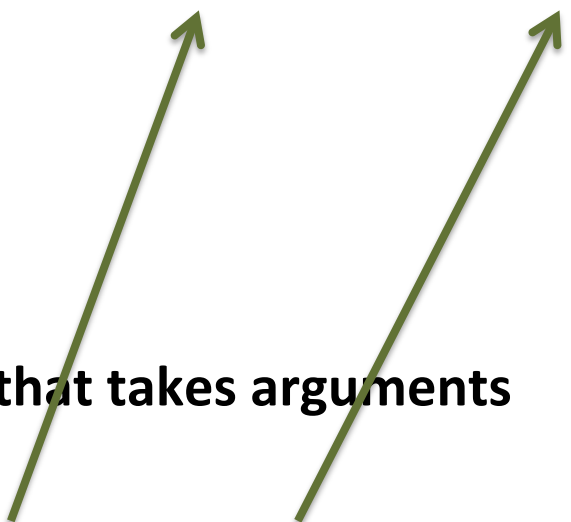
Attempting to use name here will cause an error.

- We'd like some way for `main` to be able to communicate with the `sing_song`.
- Specifically, we'd like a way for `main` to "send" the value of the variable name to `sing_song` so `sing_song` may use it.

Passing information via arguments

Syntax for defining a function that takes arguments

```
def name_of_function(variable1, variable2, ...):  
    statement  
    statement  
    statement
```



Syntax for calling a function that takes arguments

```
name_of_function(value1, value2, ...)
```


Passing information via arguments

Imagine extra assignment statements in the function body:

```
def name_of_function(variable1, variable2, ...):  
    variable1 = value1           # Python does this  
    variable2 = value2         # behind the scenes.  
  
    statement  
    statement  
    more statements...
```

The values (value1, value2, etc) come from where the function is called.

```
def sing_song(name, age):  
    print("Happy birthday to you! \  
        Happy birthday to you!")  
    print("Happy birthday dear", name, \  
        "Happy birthday to you!")  
    print("You are now", age, "years old!")
```

```
def main():  
    sing_song("Brian", 84)  
    sing_song("Meg", 27)
```

```
main()
```

```
def sing_song(name, age):  
    print("Happy birthday to you! \\  
        Happy birthday to you!")  
    print("Happy birthday dear", name, \  
        "Happy birthday to you!")  
    print("You are now" age, "years old!")
```

```
def main():  
    sing_song("Brian", 84)  
    sing_song("Meg", 27)
```

```
main()
```

When Python runs this line of code, it

- assigns "Brian" to `sing_song`'s variable name
- assigns 84 to `sing_song`'s variable age
- Runs the body of `sing_song`

```
def sing_song(name, age):  
    print("Happy birthday to you! \  
        Happy birthday to you!")  
    print("Happy birthday dear", name, \  
        "Happy birthday to you!")  
    print("You are now", age, "years old!")
```

```
def main():  
    sing_song("Brian", 84)  
    sing_song("Meg", 27)
```

```
main()
```

When Python runs this line of code, it

- assigns "Meg" to `sing_song`'s variable name
- assigns 27 to `sing_song`'s variable age
- Runs the body of `sing_song`

Output:

Happy birthday to you! Happy birthday to you!
Happy birthday dear **Brian** Happy birthday to you!
You are now **84** years old!
Happy birthday to you! Happy birthday to you!
Happy birthday dear **Meg** Happy birthday to you!
You are now **27** years old!

```
def sing_song(name, age):
    print("Happy birthday to you! \
          Happy birthday to you!")
    print("Happy birthday dear", name, \
          "Happy birthday to you!")
    print("You are now", age, "years old!")

def main():
    username = input("What is your name? ")
    their_age = int(input("What is your age? "))
    sing_song(username, their_age)
```

main()

When Python runs the green line of code, it

- assigns the value of main's variable `username` to `sing_song`'s variable `name`
- assigns the value of main's variable `their_age` to `sing_song`'s variable `age`
- Runs the body of `sing_song`

```
def sing_song(name, age):
    print("Happy birthday to you! \
          Happy birthday to you!")
    print("Happy birthday dear", name, \
          "Happy birthday to you!")
    print("You are now", age, "years old!")

def main():
    name = input("What is your name? ")
    age = int(input("What is your age? "))
    sing_song(name, age)
```

main()

- You *may* use the same variable names in both places, if desired.
- However, each function then has its own copy of the variables (4 variables total in this code, not 2!)
- Using the same variable name in both places does not "link" the variables.
- The transfer is still one-way only!

```
def some_function(x):  
    print("Inside the function, x is", x)  
    x = 17  
    print("Inside the function, x is changed to", x)  
  
def main():  
    x = 2  
    print("Before the function call, x is", x)  
    some_function(x)  
    print("After the function call, x is", x)
```

```
main()
```

Output:

```
Before the function call, x is 2  
Inside the function, x is 2  
Inside the function, x is 17  
After the function call, x is 2
```


Wait. What?

- There is no permanent connection between the `x` in `main` and the `x` in `some_function`.
- Arguments are passed --- one way only --- from `main` to `some_function` when `main` calls `some_function`.
 - This copies `main`'s value of `x` into `some_function`'s `x`.
- Any assignments to `x` inside of `some_function` do not come back to `main`.

- *"That sounds like local variables."*
- Yes, just as local variables are invisible outside of the functions that own them, variables used as arguments inside a function definition are local to that function.
- So arguments in a function definition are really local variables that happen to have some hidden assignment statements at the beginning.

```
def sing_song(name, age):
    print("Happy birthday to you! \
          Happy birthday to you!")
    print("Happy birthday dear", name, \
          "Happy birthday to you!")
    print("You are now", age, "years old!")
    age_next_year = age + 1

def main():
    sing_song("Brian", 84)
    print("Next year you will be", age_next_year, \
          "years old!")
```

main()

- This program crashes because `age_next_year` is a local variable to `sing_song`!
- `sing_song` can't send any variables back to `main`.

Tricky example

```
def mystery(x, z, y):  
    print(z, y-x)  
  
def main():  
    x = 9  
    y = 2  
    z = 5  
    mystery(z, y, x)  
    mystery(y, x, z)  
    mystery(x + z, y - x, y)  
  
main()
```

- Write a program to compute the average of three numbers typed in from the keyboard.
- Your main() function should ask the user for the numbers by using input().
- Then write an average() function that takes three arguments and prints the average of the arguments.
- If you finish early, expand your robot routine program from Wednesday to include arguments.
 - For instance, if you have a eat_meal function, you could add an argument that tells what meal of the day it is, so you could call eat_meal("breakfast") or eat_meal("lunch").