# CMSC201
# Computer Science I for Majors

## Lecture 09 – Functions

## Last Class We Covered

- The string data type
  - Built-in functions
- Miscellaneous details
  - Constants
  - Boolean flags in **`while`** loops

## Any Questions from Last Time?

## Today's Objectives

- To learn why you would want to divide your code into smaller, more specific pieces (functions!)
- To be able to define new functions in Python
- To understand the details of function calls and parameter passing in Python
- To use functions to reduce code duplication and increase program modularity

## Control Structures (Review)

- A program can proceed:
  - In sequence
  - Selectively (branching): make a choice
  - Repetitively (iteratively): looping
  - By calling a function

focus of today's lecture

## Introduction to Functions

## Functions We’ve Seen

- We’ve actually seen (and used) two different types of functions already!
- Built-in Python functions
  - For example: **`print()`**, **`input()`**, casting, etc.
- Our program’s code is contained completely inside the **`main()`** function
  - A function that we created ourselves

## Parts of a Function

use "**def**" to create a function

```
def main():
    a = 5
    print(a)
    print(type(a))
main()
```

function body

calls "**print**" function

calls "**type**" function

calls "**main**" function

## Why Use Functions?

- Functions reduce code duplication and make programs more easy to understand and maintain
- Having identical (or similar) code in more than one place has various downsides:
  1. Have to write the same code twice (or more)
  2. Must be maintained in multiple places
  3. Hard to understand big blocks of code everywhere

## What are Functions?

- A ***function*** is like a subprogram
  - A small program inside of a program
- The basic idea:
  - We write a sequence of statements
  - And give that sequence a ***<u>name</u>***
  - We can then execute this sequence at any time by referring to the sequence's name

## When to Use Functions?

- Functions are used when you have a block of code that you want to be able to:
  - Write once and be able to use again
    - Example: getting input from the user
  - Call multiple times at different places
    - Example: printing out a menu of choices
  - Change a little bit when you call it each time
    - Example: printing out a greeting to different people

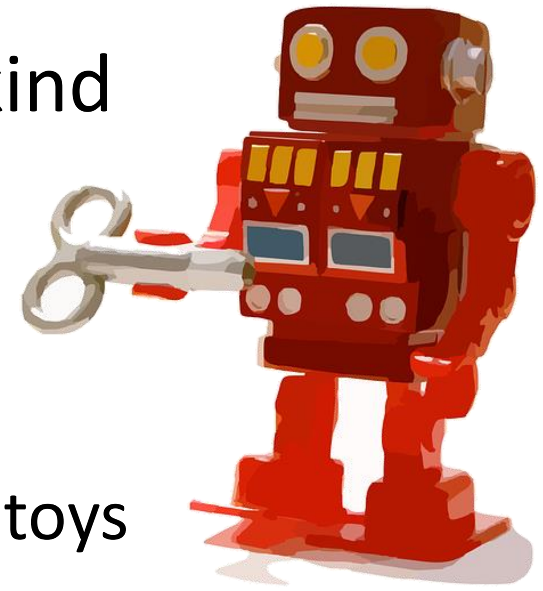
# Function Vocabulary

- Function **definition**:
  - The part of the program that creates a function
  - For example: “`def main():`” and the lines of code that are indented inside of `def main():`
- Function **call**:
  - When the function is used in a program
  - For example: “`main()`” or “`print("Hello")`”

## Function Example

## Note: Toy Examples

- The example we're going to look at today is something called a ***toy example***
- It is purposefully simplistic (and kind of pointless) so you can focus on:
  - The concept being taught
  - Not how the code itself works
- Sadly, it has nothing to do with actual toys



## "Happy Birthday" Program

- Happy Birthday lyrics…

```
def main():
    print("Happy birthday to you!")
    print("Happy birthday to you!")
    print("Happy birthday, dear Maya...")
    print("Happy birthday to you!")
main()
```

- Gives us this…

```
bash-4.1$ python birthday.py
Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Maya...
Happy birthday to you!
```

## Simplifying with Functions

- Most of this code is repeated (duplicate code)

```
print("Happy birthday to you!")
```

- We can ***define*** a function to print out that line

```
def happy():
    print("Happy birthday to you!")
```

- Let’s update our program to use this function

## Updated "Happy Birthday" Program

- The updated program:

```
def happy():
    print("Happy birthday to you!")


def main():
    happy()
    happy()
    print("Happy birthday, dear Maya...")
    happy()
main()
```

## More Simplifying

- This clutters up our main function, though
- We could write a separate function that sings “Happy Birthday” to Maya, and call it in **`main()`**

```
def singMaya():
    happy()
    happy()
    print("Happy birthday, dear Maya...")
    happy()
```

# New Updated Program

- The new updated program:

```
def happy():
    print("Happy birthday to you!")
def singMaya():
    happy()
    happy()
    print("Happy birthday, dear Maya...")
    happy()
def main():
    singMaya() # sing Happy Birthday to Maya
main()
```

## Updated Program Output

```
bash-4.1$ python birthday.py
Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Maya...
Happy birthday to you!
```

Notice that despite all the changes we made to the code, the output is still exactly the same as before

## Someone Else's Birthday

- Creating this function saved us a lot of typing!
- What if it's Luke's birthday?
  - We could write a new **`singLuke()`** function!

```
def singLuke():
    happy()
    happy()
    print("Happy birthday, dear Luke...")
    happy()
```

## “Happy Birthday” Functions

```
def happy():
    print("Happy birthday to you!")
def singMaya():
    happy()
    happy()
    print("Happy birthday, dear Maya...")
    happy()
def singLuke():
    happy()
    happy()
    print("Happy birthday, dear Luke...")
    happy()
def main():
    singMaya() # sing Happy Birthday to Maya
    print()    # empty line between the two (take a breath!)
    singLuke() # sing Happy Birthday to Luke
main()
```

## Updated Program Output

```
bash-4.1$ python birthday2.py
Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Maya...
Happy birthday to you!

Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Luke...
Happy birthday to you!
```

Image from pixabay.com

## Multiple Birthdays

- This is much easier to read and use!
- But… there's still a lot of code duplication

- The only difference between **`singMaya()`** and **`singLuke()`** is what?
  - The name in the third **`print()`** statement
- We could combine these two functions into one by using something called a ***parameter***

## Function Parameters

## What is a Parameter?

- A ***parameter*** is a variable that is initialized when we call a function
- We can create a **`sing()`** function that takes in a person's name (a string) as a parameter

```
def sing(name):
    happy()
    happy()
    print("Happy birthday, dear", name, "...")
    happy()
```

parameter

## “Happy Birthday” with Parameters

```
def happy():
    print("Happy birthday to you!")

def sing(name):
    happy()
    happy()
    print("Happy birthday, dear", name, "...")
    happy()

def main():
    sing("Maya")
    print()
    sing("Luke")
main()
```

# "Happy Birthday" with Parameters

```
def happy():
    print("Happy birthday to you!")

def sing(name):
    happy()
    happy()
    print("Happy birthday, dear", name, "...")
    happy()

def main():
    sing("Maya")
    print()
    sing("Luke")
main()
```

parameter passed in

parameter being used

function call with parameter

function call with parameter

## Updated Program Output

```
bash-4.1$ python birthday3.py
Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Maya...
Happy birthday to you!

Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Luke...
Happy birthday to you!
```

This looks the same as before!

That's fine! We wanted to make our code easier to read and use, not change the way it works.

## Exercise: Prompt for Name

- How would we update the code in **`main()`** to ask the user for the name of the person?
  - Current code looks like this:

```
def main():
    sing("Maya")
main()
```

## Solution: Prompt for Name

- How would we update the code in **`main()`** to ask the user for the name of the person?
  - Updated code looks like this:

```
def main():
    birthdayName = input("Whose birthday? ")
    sing(birthdayName)
main()
```

Nothing else needs to change – and the **`sing()`** function stays the same

## Exercise Output

```
bash-4.1$ python birthday4.py
Whose birthday? UMBC
Happy birthday to you!
Happy birthday to you!
Happy birthday, dear UMBC...
Happy birthday to you!
```

## How Parameters Work

## Functions and Parameters

- Each function is its own little subprogram
  - Variables used inside of a function are ***local*** to that function
  - Even if they have the same name as variables that appear outside that function
- The <u>**only**</u> way for a function to see a variable from outside itself is for that variable to be passed as a ***parameter***

## Function Syntax with Parameters

- A function definition looks like this:

function name: follows same syntax rules as variable names

(no special characters, can't start with a number, no keywords, etc.)

```
def fxnName(formalParameters):
    # body of the function
```

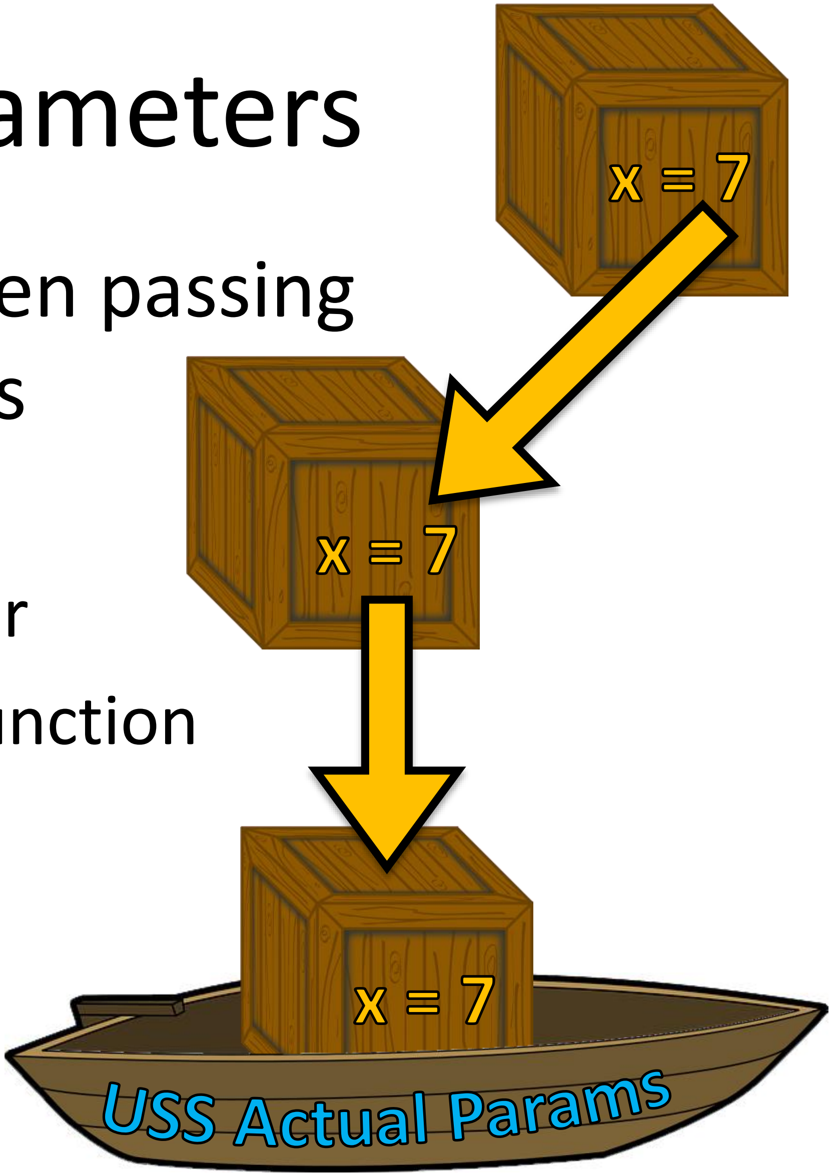
the formal parameters that the function takes in – **can be empty!**

## Formal Parameters

- The ***formal parameters***, like all variables used in the function, are <u>**only**</u> accessible in the body of the function
- Variables with identical names elsewhere in the program are distinct from those inside the function body
  - We call this the "***scope***" of a variable

## Scope: Passing Parameters

- If variables are boxes, then passing actual parameters entails
  - Making a new box
  - Copying the contents over
  - Sending it to the called function



## Scope: Passing Parameters

- If variables are boxes, then passing actual parameters entails
  - Making a new box
  - Copying the contents over
  - Sending it to the called function
- Each function is its own separate desert island, with its own variables (boxes that can hold unique values)

## Example of Scope

- This is our president, Freeman A. Hrabowski III
  - According to Wikipedia, he is a “a prominent American educator, advocate, and mathematician” and has been the President of UMBC since 1992
  - He will also take you up to the roof of the Admin building to show off the campus (it’s super cool)

## Example of Scope

- This is my (fictional) dog, a Chesapeake Bay Retriever also named Hrabowski
  - He is super cute, can "sit" and "fetch," and his favorite toy is a squeaky yellow duck
  - He also loves to spin in circles while chasing his tail

## Example of Scope

- We have two very different things, both of which are called Hrabowski:
  - UMBC’s President Hrabowski
  - My (fictional) dog Hrabowski
- If you go outside this classroom and tell someone “Hrabowski loves to chase his tail, it’s super cute” they will be <u>very</u> confused

## Example of Scope

- In the same way, a variable called **`name`** inside the function **`sing()`** is a completely <u>different</u> variable from **`name`** in **`main()`**

- The **`sing()`** function has one idea of what the **`name`** variable is, and **`main()`** has another
- It depends on the context, or "scope" we are in

## Calling Functions with Parameters

## Calling with Parameters

- In order to call a function with parameters, use its name followed by a list of variables

```
myFunction("my string", numVar)
```

- These variables are the ***actual parameters***, or ***arguments***, that are passed to the function

# Code Trace: Parameters

```
def happy():
    print("Happy birthday to you!")
def sing(name):
    happy()
    happy()
    print("Happy birthday, dear", name, "...")
    happy()

def main():
    sing("Maya")
    print()
    sing("Luke")

main()
```

formal parameter

actual parameter

actual parameter

## Python and Function Calls

- When Python comes to a function call, it initiates a four-step process:
    1. The calling program ***suspends execution*** at the point of the *call*
    2. The ***formal parameters*** of the function get assigned the values supplied by the ***actual parameters*** in the call
    3. The body of the function is ***executed***
    4. ***Control*** is returned to the point <u>just after</u> where the function was called

## Code Trace: Parameters

- Let’s trace through the following code:

```
sing("Maya")
print()
sing("Luke")
```

- When Python gets to the line `sing("Maya")`, execution of **`main`** is temporarily suspended
- Python looks up the definition of **`sing()`** and sees it has one formal parameter, **`name`**

## Initializing Formal Parameters

- The ***formal parameter*** is assigned the value of the ***actual parameter***

- When we call `sing("Maya")`, it as if the following statement was executed in `sing()`

```
    name = "Maya"
```

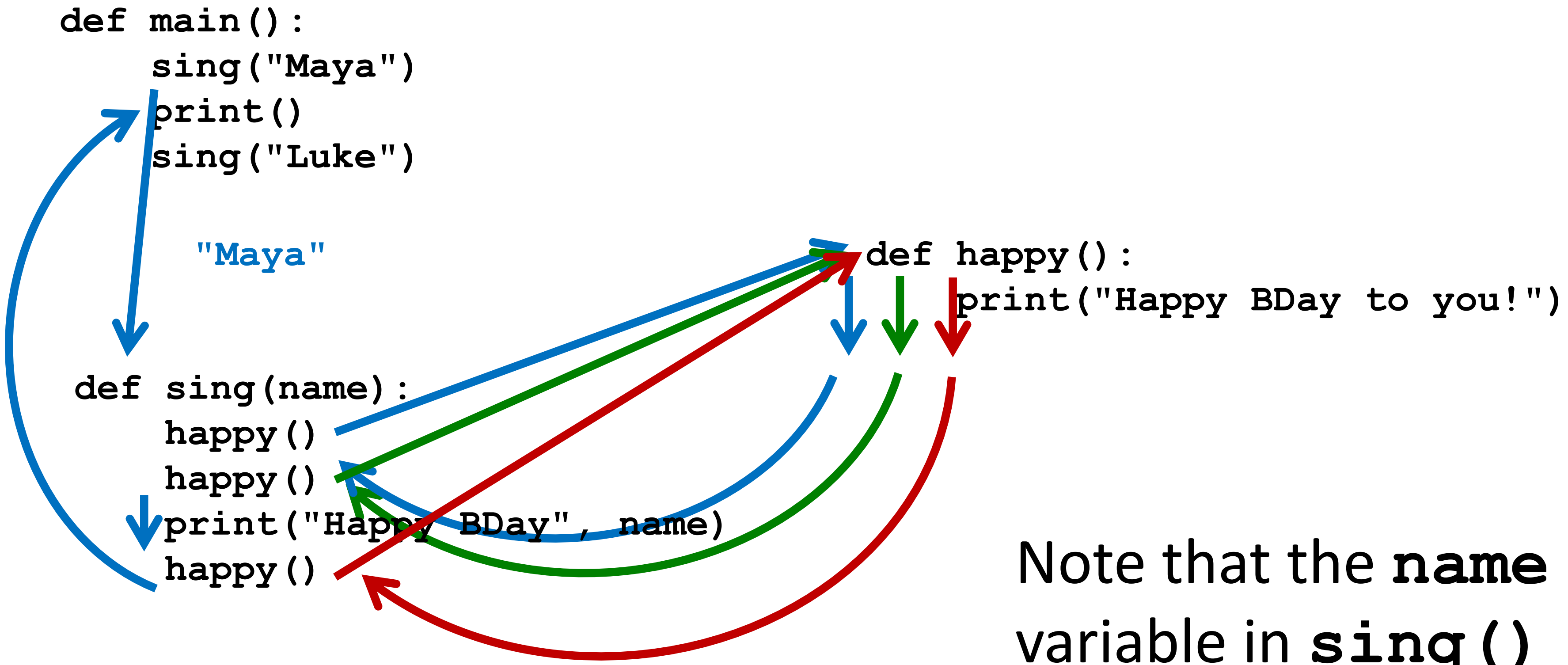
## Code Trace: Parameters

- Next, Python begins executing the body of the **`sing()`** function
  - First statement is another function call, to **`happy()`** – what does Python do now?
    - Python suspends the execution of **`sing()`** and transfers control to **`happy()`**
    - The **`happy()`** function's body is a single **`print()`** statement, which is executed
  - Control returns to where it left off in **`sing()`**

## Code Trace: Parameters

- Execution continues in this way with two more “trips” to the **`happy()`** function
- When Python gets to the end of **`sing()`**, control returns to…
  - **`main()`**, which picks up…
  - where it left off, on the line immediately following the function call

## Visual Code Trace



```
def main():
    sing("Maya")
    print()
    sing("Luke")
```

```
def sing(name):
    happy()
    happy()
    print("Happy BDay", name)
    happy()
```

```
def happy():
    print("Happy BDay to you!")
```

Note that the **`name`** variable in **`sing()`** disappeared after we exited the function!

## Local Variables

- When a function exits, the local variables (like **`name`**) are deleted from memory

- If we call **`sing()`** again, a new **`name`** variable will have to be re-initialized
  - Local variables do **not** retain their value between function calls

## Code Trace: Parameters

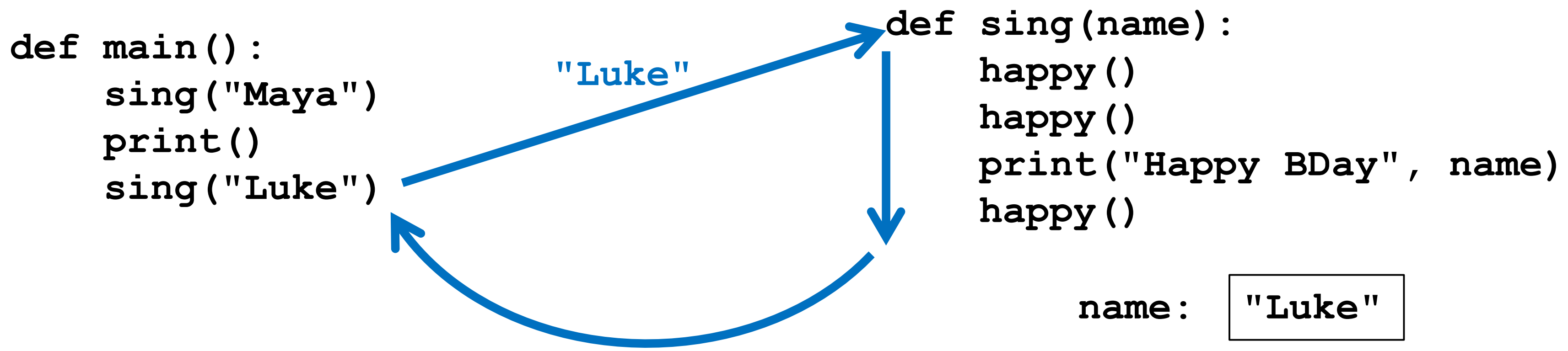
- Next statement in **`main()`** is the empty call to **`print()`**, which simply produces a blank line

- Python sees another call to **`sing()`**, so…
  - It suspends execution of **`main()`**, and…
  - Control transfers to…

    the **`sing()`** function
  - With the actual parameter…

    **`"Luke"`**

# Visual Code Trace



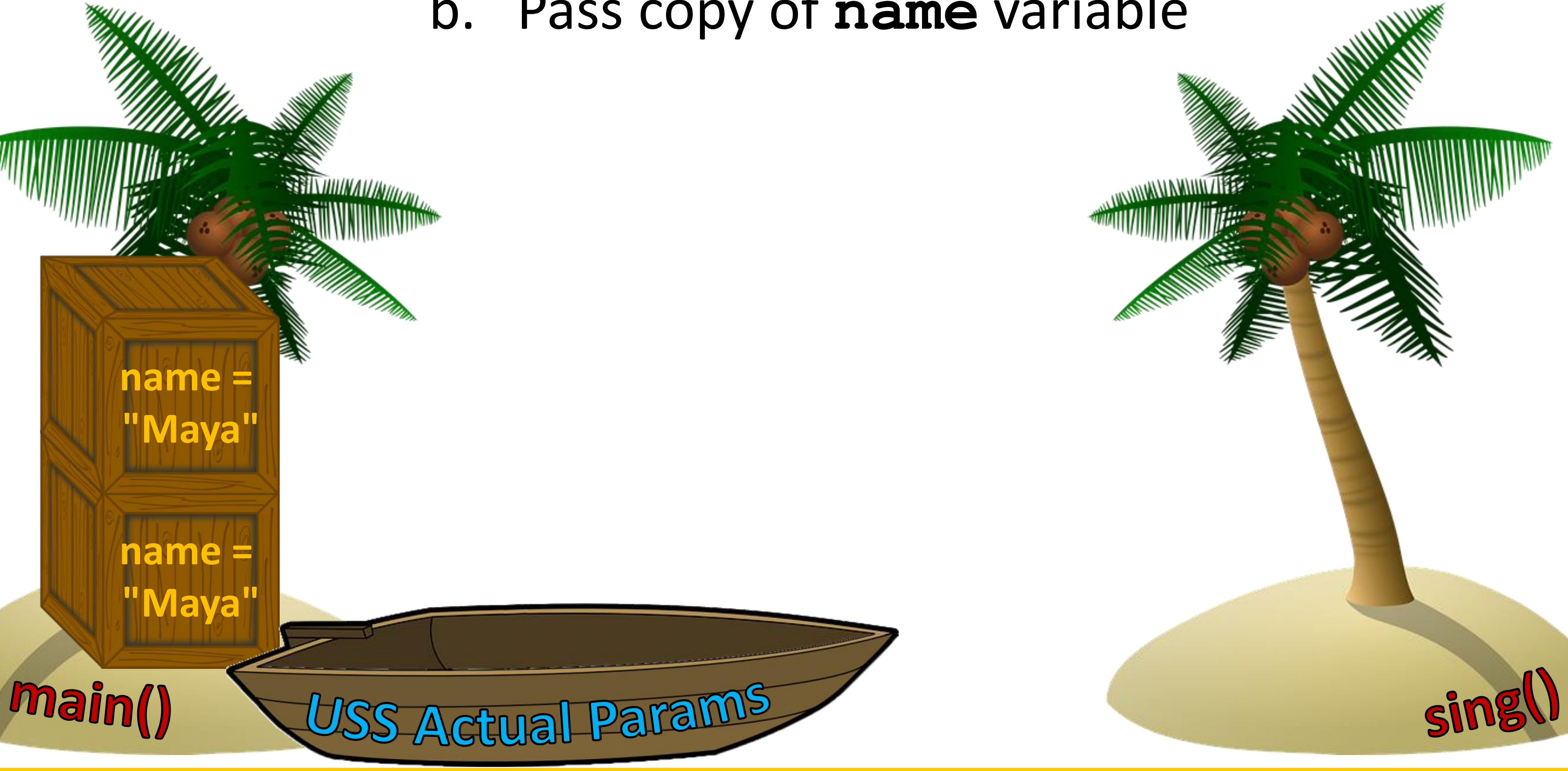
```
def main():
    sing("Maya")
    print()
    sing("Luke")
```

```
def sing(name):
    happy()
    happy()
    print("Happy BDay", name)
    happy()
```

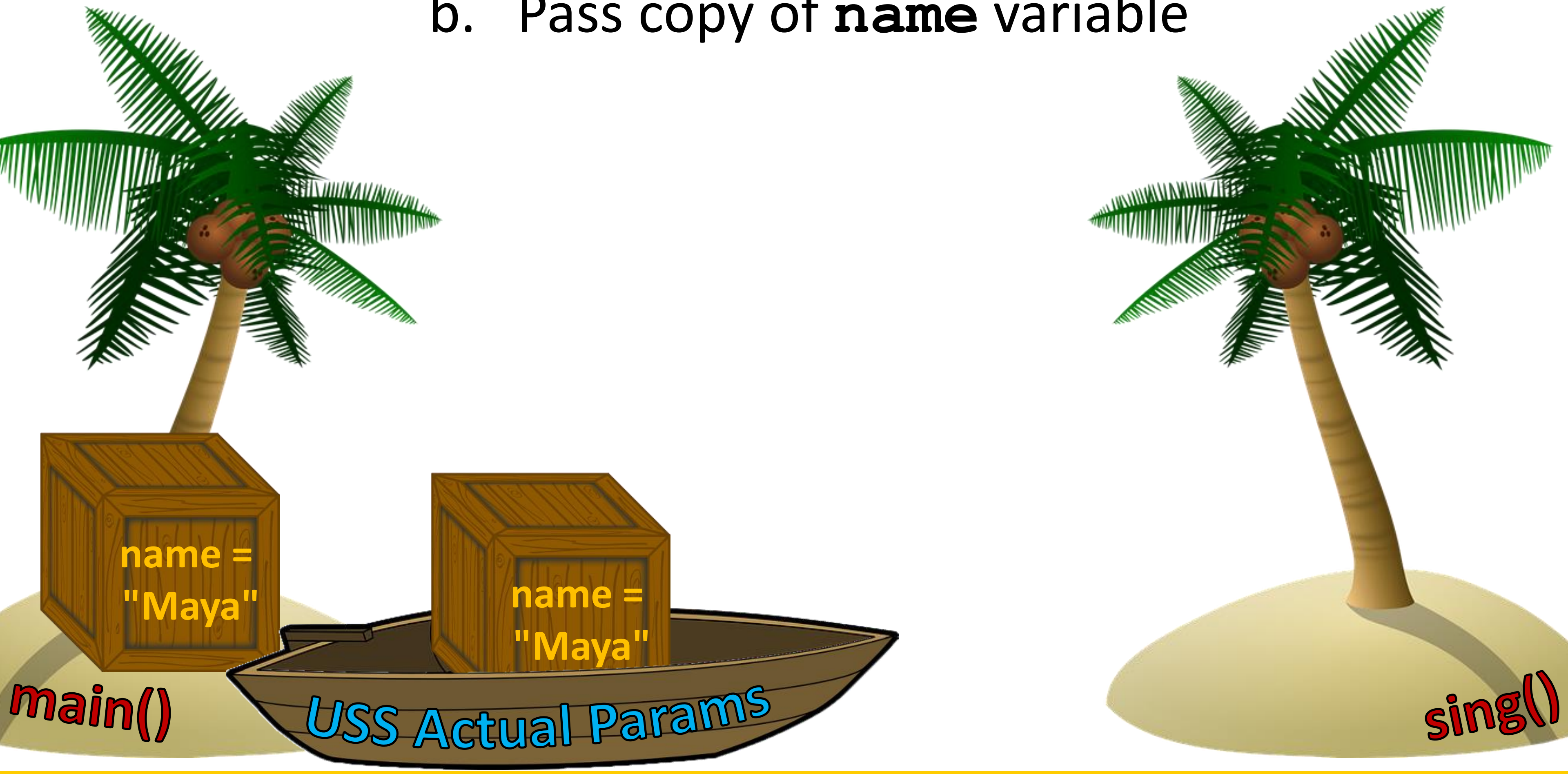
- The body of **`sing()`** is executed with the argument **`"Luke"`**
  - Including its three side trips to **`happy()`**
- Control then returns to **`main()`**

## Island Example

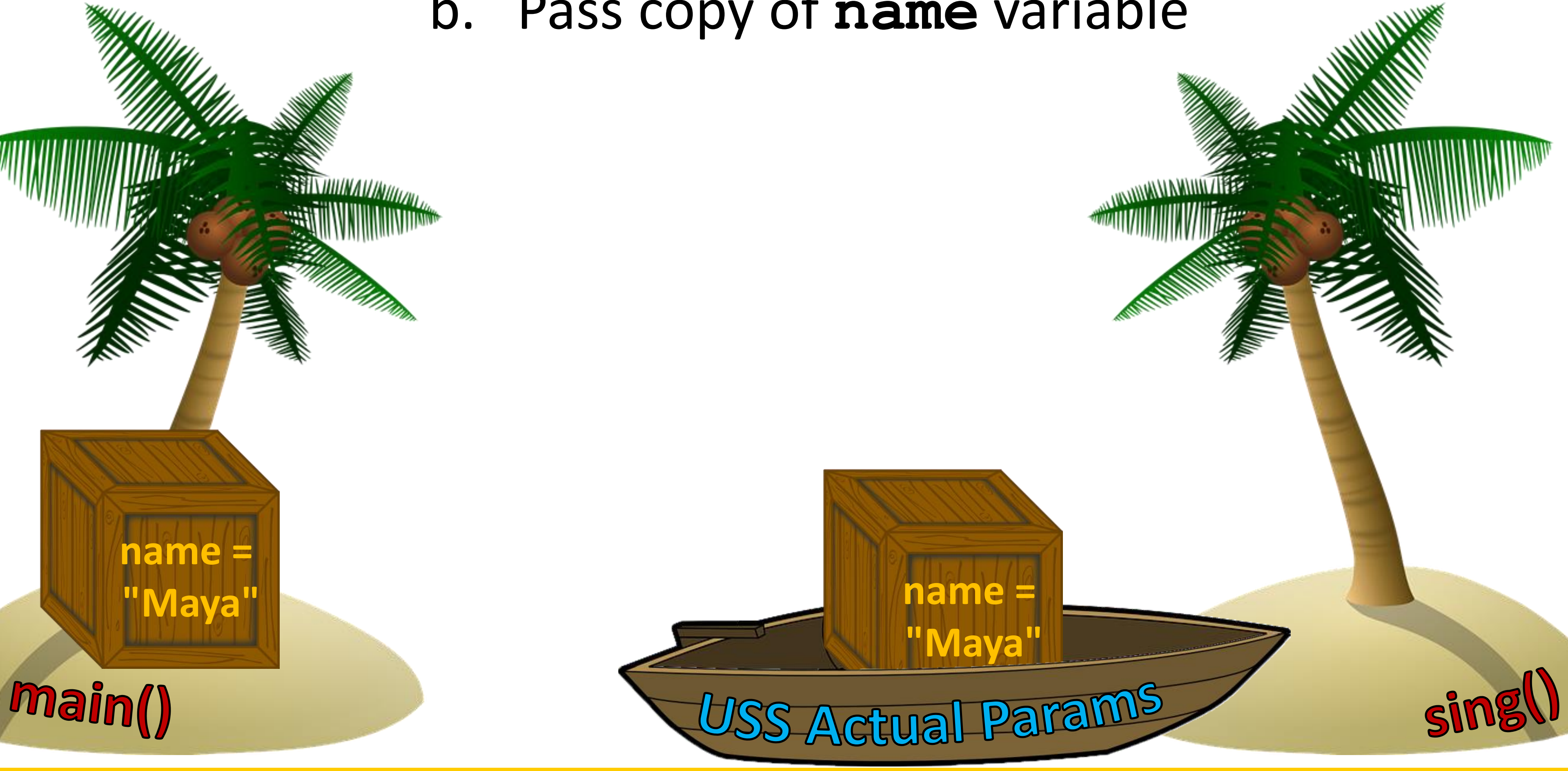
1. Function **`sing()`** is called
    a. Make copy of **`name`** variable
    b. Pass copy of **`name`** variable



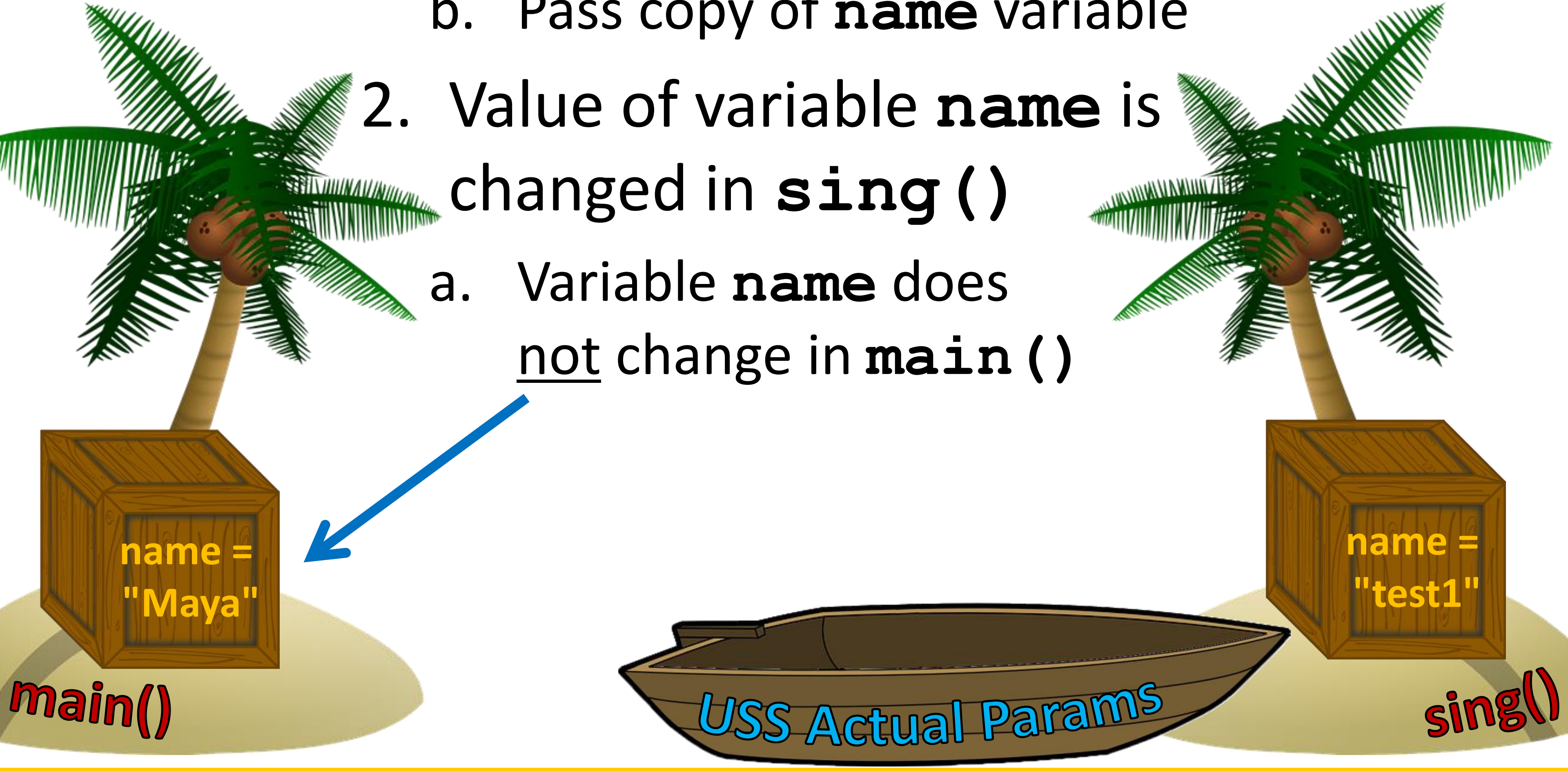
1. Function **`sing()`** is called
   a. Make copy of **`name`** variable
   b. Pass copy of **`name`** variable



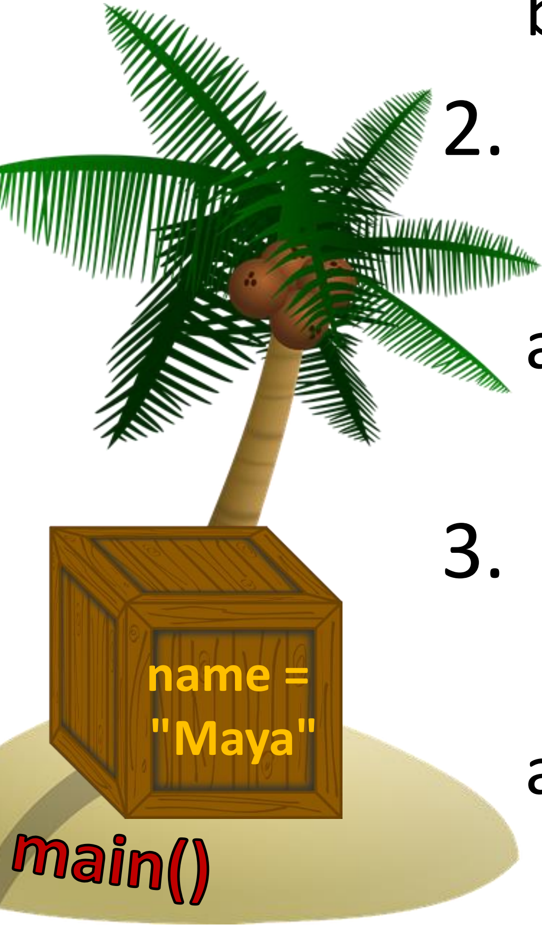
1. Function **`sing()`** is called
    a. Make copy of **`name`** variable
    b. Pass copy of **`name`** variable

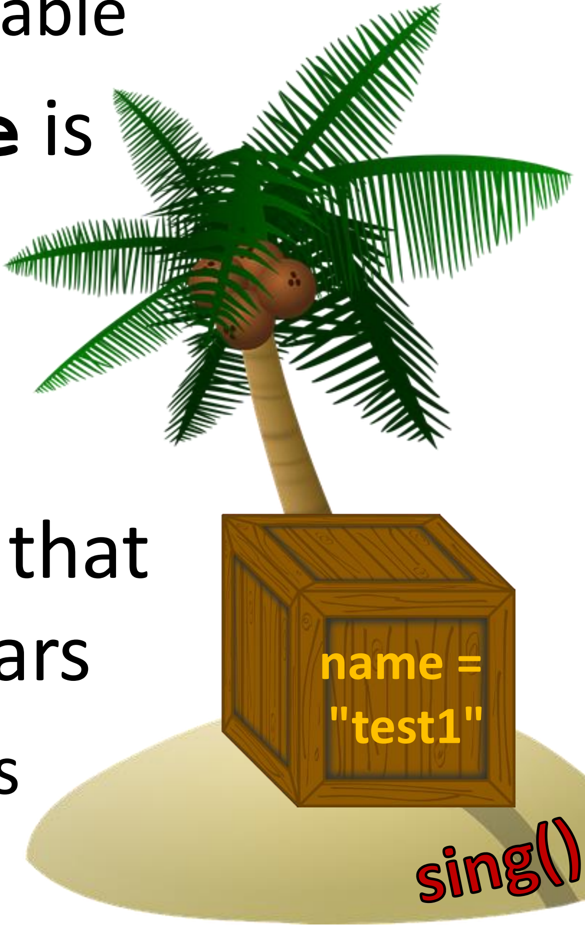


1. Function **`sing()`** is called
   a. Make copy of **`name`** variable
   b. Pass copy of **`name`** variable
2. Value of variable **`name`** is changed in **`sing()`**
   a. Variable **`name`** does not change in **`main()`**



1. Function **`sing()`** is called
   a. Make copy of **`name`** variable
   b. Pass copy of **`name`** variable
2. Value of variable **`name`** is changed in **`sing()`**
   a. Variable **`name`** does <u>not</u> change in **`main()`**
3. When **`sing()`** exits, that copy of **`name`** disappears
   a. And any other variables local to **`sing()`**





## Multiple Parameters

## Multiple Parameters

- One thing we haven't discussed is functions with ***multiple parameters***

- When a function has more than one parameter, the formal and actual parameters are matched up based on **<u>position</u>**
  - First actual parameter becomes the first formal parameter, etc.

## Multiple Parameters in **`sing()`**

- Let's add a second parameter to **`sing()`** that will take in the person's age as well
- And print out their age in the song

```
def sing(name, age):
    happy()
    happy()
    print("Happy birthday, dear", name, "...")
    print("You're", age, "years old now...")
    happy()
```

## Multiple Parameters in **`sing()`**

- What will happen if we use the following call to the **`sing()`** function in **`main()`**?

```
def main():
    sing("Maya", 7)
main()
```

- It will print out:

```
Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Maya...
You're 7 years old now...
Happy birthday to you!
```

## Assigning Parameters

- Python is simply assigning the first actual argument to the first formal argument, etc.

```
sing("Maya", 7) # function call



def sing(name, age):
    # function body goes here
```

## Parameters Out-of-Order

- What will happen if we use the following call to the **`sing()`** function in **`main()`**?

```
def main():
    sing(7, "Maya")
main()
```

- It will print out:

```
Happy birthday to you!
Happy birthday to you!
Happy birthday, dear 7...
You're Maya years old now...
Happy birthday to you!
```

## Parameters Out-of-Order

- Python isn't smart enough to figure out what you meant for your code to do
  - It only understands the exact code

- That's why it matches up actual and formal parameters based only on their order

## Announcements

- HW 4 is out on Blackboard now
  - All assignments will be available only on Blackboard until after the due date
  - Complete the Academic Integrity Quiz to see it
  - Due by Friday (March 3rd) at 8:59:59 PM
- Midterm is <u>in class</u>, March 15th and 16th
  - Week before Spring Break
  - Survey #1 will be released that week as well