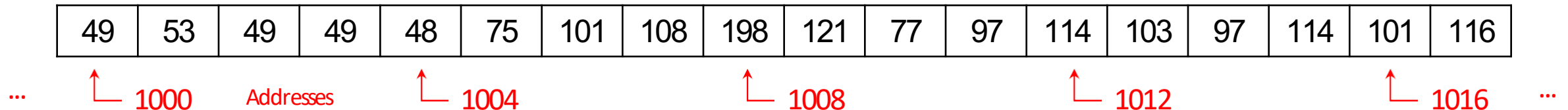


# Functions

# Computer Memory is Stored as Binary

Your computer keeps track of saved data and all the information it needs to run in its memory, which is represented as binary. You can think about your computer's memory as a really long list of bits, where each bit can be set to 0 or 1. But usually we think in terms of bytes, groups of 8 bits.

Every byte in your computer has an address, which the computer uses to look up its value.



# Binary Values Depend on Interpretation

When you open a file on your computer, the application goes to the appropriate address, reads the associated binary, and interprets the binary values based on the file encoding it expects. That interpretation depends on the application you use when opening the file, and the filetype.

You can attempt to open any file using any program, if you convince your computer to let you try. Some programs may crash, and others will show nonsense because the binary isn't being interpreted correctly.

Example: try changing a .docx filetype to .txt, then open it in a plain text editor. .docx files have extra encoding, whereas .txt files use plain ASCII.

# Learning Objectives

- Identify the argument(s), returned value, and side effect(s) of a function call
- Use function definitions when reading and writing algorithms to implement procedures that can be repeated on different inputs
- Recognize the difference between local and global scope

# Repeating Actions is Messy

Sometimes we want to perform the same algorithm many times on different inputs.

For example, say we want to personalize a young child's reading material so that it uses their pet's name.

We could copy and paste the first bit of code, then change the necessary parts. But if we're sloppy this might cause errors.

```
pet1 = "Spot"  
pet2 = "Stella"  
pet3 = "Kimchee"
```

```
print("See " + pet1 + ". See " + pet1 +  
      " run. Run, " + pet1 + ", run!")
```

```
print("See " + pet2 + ". See " + pet2 +  
      " run. Run, " + pet2 + ", run!")
```

```
print("See " + pet3 + ". See " + pet1 +  
      " run. Run, " + pet3 + ", run!")
```

# Functions Represent Abstract Actions

A better approach is to put the core action being repeated into a function.

A function is a code construct that represents an algorithm. We can define a function once, then call it many times.

We can also use functions that have already been defined by Python.

# Function Calls

# Call Functions with Parentheses

We've already seen how to call a function on a specific input, because `print` is just a function! This is done using parentheses.

*functionName(input1, input2, ...)*

The number of inputs provided inside the parentheses depends on how many inputs the function expects. Each input should be an expression.



# A Few New Functions

To help us explore how functions work, let's introduce a few new functions. These are built-in functions, like `print`; that means we can call them in Python directly.

`abs(-2)` # absolute value

`pow(2, 3)` # raises a number to the given power

`round(12.4567, 2)` # rounds to the given # sig digs

# Type Functions

There are a few other built-in functions that are helpful to know, as they let you change the type of data values. This is called type-casting.

`int("4")` # converts a value to an integer

`float(3)` # converts a value to a float

`str(98.9)` # converts a value to a string

`bool(0)` # converts a value to a Boolean

`type(4 + 3.0)` # returns the type of the eventual value

# uses the names we covered before – int, float, str, bool

# Components of Functions

The functions we call may have three components:

Argument(s) – the values that are provided inside the parentheses

Returned Value – what the function evaluates to after running

Side Effect(s) – any change(s) that happen while the function runs

# Arguments Provide the Input

The specific inputs we provide to a function are called arguments. In the function call `abs(4)`, the argument is `4`.

Arguments are separated by commas and placed between the parentheses of the function call. Functions can require as many (or as few) arguments as needed.

The positions of the arguments usually have meaning. In `pow(2, 3)`, the first argument is the base and the second argument is the exponent. In other words, `pow(2, 3)` and `pow(3, 2)` mean two different things.

# Receive Output as Returned Value

When a built-in function takes its arguments and runs through its algorithm, we cannot see what it is doing.

When the function is done, it sends back an output as a returned value. We usually say a function returns a value. This value substitutes in for the function call the same way a variable's value substitutes in for the variable.

For example, the returned value of `pow(2, 3)` is 8.

# Side Effects Show Change

Python needs to keep track of certain pieces of data that change over time as a program runs (like which variables exist and what their values are, what has been printed to the screen, etc). We call this information the program state. When you set a variable to a new value, you change that program's state.

Sometimes a function changes the program state in an observable way as it is running; for example, it might display values in the interpreter, or modify a file, or produce graphics. This is called a side effect.

If we call `pow(2, 3)`, there is no observable side effect. But `print("Hello")` has an observable side effect: it prints "Hello" to the screen.

# Side Effect(s) vs Returned Value

It's easy to get confused about whether something is a side effect or a returned value. Why are these two components different?

The way we've set up function calls means that there must be exactly one output: the returned value. A function call might have no side effects, or one, or many; however, every function call has one returned value.

Importantly, returned values can be saved in a variable and used in future computations. Side effects cannot be saved; we simply observe them.

# Missing Returned Values are None

If a function produces no explicit output (usually because it is only used for side effects, like `print`), it still has a returned value. That value is the built-in value `None`.

`None` means that there was no explicit output to be returned. Like `True` and `False`, its meaning is built into Python, so it does not need quotes.

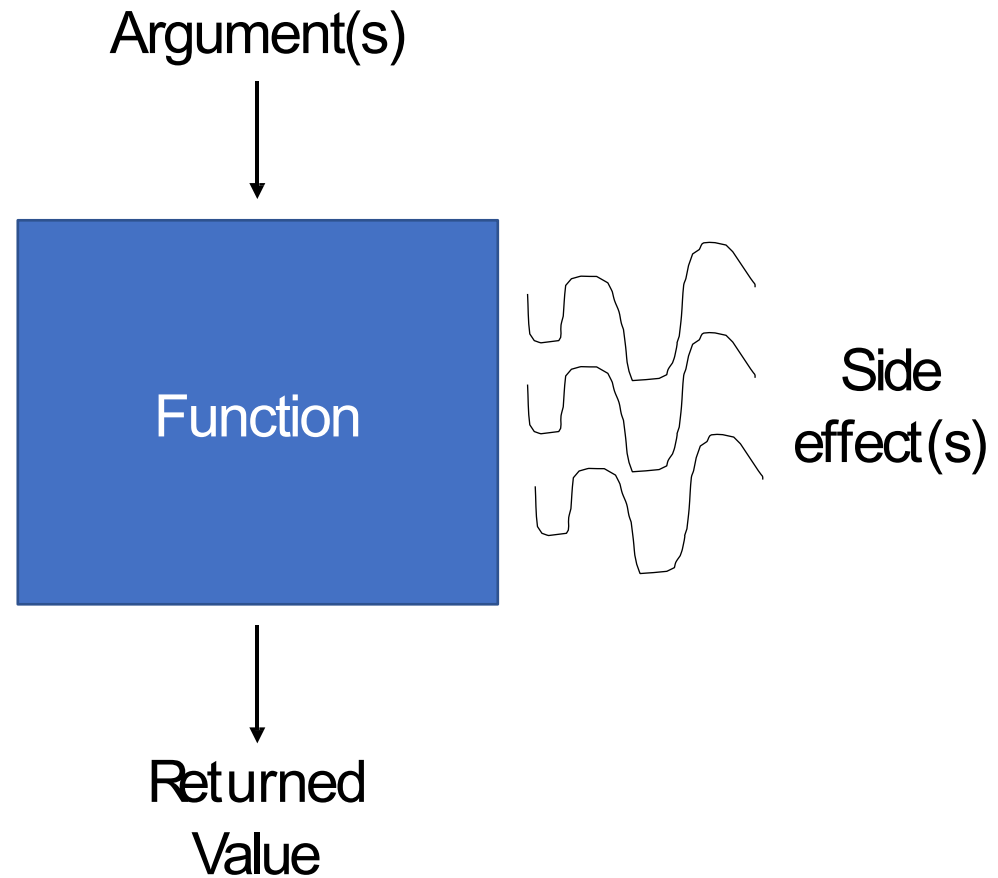
If you try to set a variable to a `print` call, you'll find that the variable holds `None`. Note that `None` does not show up in the interpreter unless you explicitly `print` it; the interpreter just shows a blank instead.

```
>>> None

>>> print(None)
None
```



# Function Call Process



# Activity – Identify the Function Call Parts

Consider the following two function calls. For each function call, what are its argument(s), returned value, and observable side effect(s)?

```
round(3.14159, 1)
```

```
print("20", "-", "22")
```

# Function Definitions

# Function Definitions Run on Abstract Input

Now that we have all the individual components of functions, we can write new function definitions ourselves.

To write a function, you need to determine what algorithm you want to implement. You'll convert that algorithm into code that runs on abstract input.

# Core Function Definition

Let's start with a simple function that has no explicit input or output; instead, it has a side effect (printed lines).

```
def helloWorld():  
    print("Hello World!")  
    print("How are you?")
```

```
helloWorld()
```

`def` is how Python knows the following code is a function definition

`helloWorld` is the name of the function. This is how we'll call it.

The colon at the end of the first line, and the indentation at the beginning of the second and third, tell Python that we're in the body of the function.

The body holds the algorithm. When the indentation stops, the function is done.

In this example, the last line calls the function we've written.

# Parameters are Abstracted Arguments

To add input to the function definition, add parameters inside the parentheses next to the name.

These parameters are variables that are not given initial values. Their initial values will be provided by the arguments given each time the function is called.

```
def hello(name):  
    print("Hello, " + name + "!!")  
    print("How are you?")
```

```
hello("Stella")  
hello("Dippy")
```

# Return Provides the Returned Value Output

To make our function have a non-`None` output, we need to have a return statement. This statement specifies the value that should be substituted for the function call when the function is called on a specific input.

```
def makeHello(name):  
    return "Hello, " + name + "! How are you?"
```

```
s = makeHello("Scotty")
```

As soon as Python returns a value, it exits the function. Python ignores any lines of code after a return statement.

# Example Side Effect Function

As in function calls, we can implement side effects in function definitions by writing code that changes the program's state. The following function call has several observable side effects and a returned value of `None`.

```
def singHappyBirthday(name):  
    print("Happy birthday to you")  
    print("Happy birthday to you")  
    print("Happy birthday dear " + name)  
    print("Happy birthday to you!")  
val = singHappyBirthday("Dippy")
```



# Example Return Function

On the other hand, this function call has a float as the returned value and no side effects.

```
def addTip(cost, percentToTip):  
    return cost + cost * percentToTip
```

```
total = addTip(20.00, 0.17)
```

# Activity: Side Effects and Returned Value

You do: what are the observable side effects and returned value of the following function call?

```
def distance(x1, y1, x2, y2):  
    xPart = (x2 - x1)**2  
    yPart = (y2 - y1)**2  
    print("Partial Work:", xPart, yPart)  
    return (xPart + yPart) ** 0.5
```

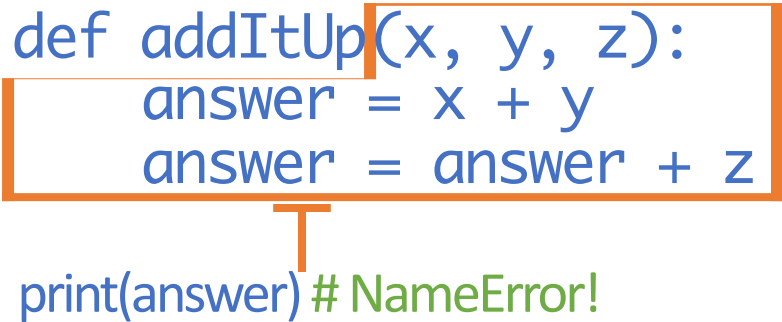
```
result = distance(0, 0, 3, 4)
```

# Scope

# Variables Have Different Scopes

All the work done in a function is only accessible in that function. In other words, if we make a variable in a function, the outer program can't access it; the only way to transmit its value is to return it.

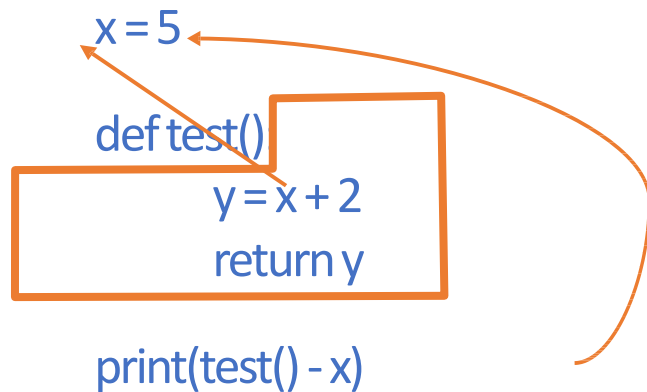
```
def addItUp(x, y, z):  
    answer = x + y  
    answer = answer + z  
  
print(answer) # NameError!
```

A diagram illustrating variable scope. A blue box highlights the function definition and its internal assignments. An orange line extends from the 'answer' variable in the function down to the 'print(answer)' statement, which is followed by a green comment '# NameError!'. This visualizes that the 'answer' variable is only defined within the function's local scope and is not accessible in the global scope where the print statement is executed.

The variable `answer` has a local scope and is accessible only within the function `addItUp`.

# Everything Can Access Global Variables

On the other hand, if a function is told to use a variable it hasn't defined, the function automatically looks in the global scope (outside the function at the top level) to see if the variable exists there.



The diagram illustrates how Python resolves a variable reference. It shows a global variable `x = 5` at the top. Below it is a function definition `def test():` followed by an indented block containing `y = x + 2` and `return y`. This function block is enclosed in an orange rectangle. At the bottom is a function call `print(test() - x)`. An orange arrow originates from the `x` in `test() - x` and points to the `x = 5` line, indicating that the function looks for the variable in the global scope when it is not found in its local scope.

```
x = 5  
  
def test():  
    y = x + 2  
    return y  
  
print(test() - x)
```

If you change a global variable in a function, that's a side effect! It's unlikely that you'll want to use this, but good to know for debugging.

# Scope is Like Names

You can think of the scope of a variable as being like its last name. For example, consider the following code:

```
x = "bar"
```

```
def test():
```

```
    x = "foo"
```

```
    print("A", x)
```

```
test()
```

```
print("B", x)
```

`x` exists in both the local and the global scope, but the two `x` variables are separate and have different values.

Analogy: knowing two people both named Andrew. They have the same first name, but different last names.

In the code above, the last name of the function's `x` would be test, while the last name of the top-level `x` would be global.

In general, it's best to keep variable names different to avoid confusion.

# Activity: Local or Global?

Which variables in the following code snippet are global? Which are local?  
For the local variables, which function can see them?

```
name = "Farnam"

def greet(day):
    punctuation = "!"
    print("Hello, " + name + punctuation)
    print("Today is " + day + punctuation)

def leave():
    punctuation = "."
    print("Goodbye, " + name + punctuation)

greet("Wednesday")
leave()
```