# CSCI 141

Lecture 14
Functions:
Parameters, Local Variables, Scope, Return Value

# Announcements

# Announcements

- Midterm grading is underway

# Announcements

- Midterm grading is underway
- Lots of new syntax and concepts happening this week.

# Announcements

- Midterm grading is underway
- Lots of new syntax and concepts happening this week.
- Read Chapter 6 of the ebook and make sure you understand everything on the the Lab 5 handout.

# Announcements

- Midterm grading is underway
- Lots of new syntax and concepts happening this week.
- Read Chapter 6 of the ebook and make sure you understand everything on the the Lab 5 handout.
- You will be responsible for material I don’t cover in class, but does appear in Chapter 6 or Lab 5.

# Goals

- Know the syntax for defining your own functions
- Know how to define and use functions that take no arguments and return no values
- Know how to use parameters to refer to the input arguments in a function definition
- Know the syntax for triple-quoted strings, and how they are used to write docstrings that describe a function's specification.
- Know what does and does not belong in a function specification (see Lab 5)
- Know the meaning of local variables and variable scope and how it relates to function parameters.
- Know how to return a value from a function.

# Functions, Revisited

What **is** a function, anyway?

- As a user, you can treat a function as a **"black box":** all you need to know is:
  - the **inputs**, **effects**, and **return value**.
- Functions are named chunks of code.


A bunch of (complicated) stuff is wrapped up in a nice, easy-to-use package.

# Writing Functions: Syntax

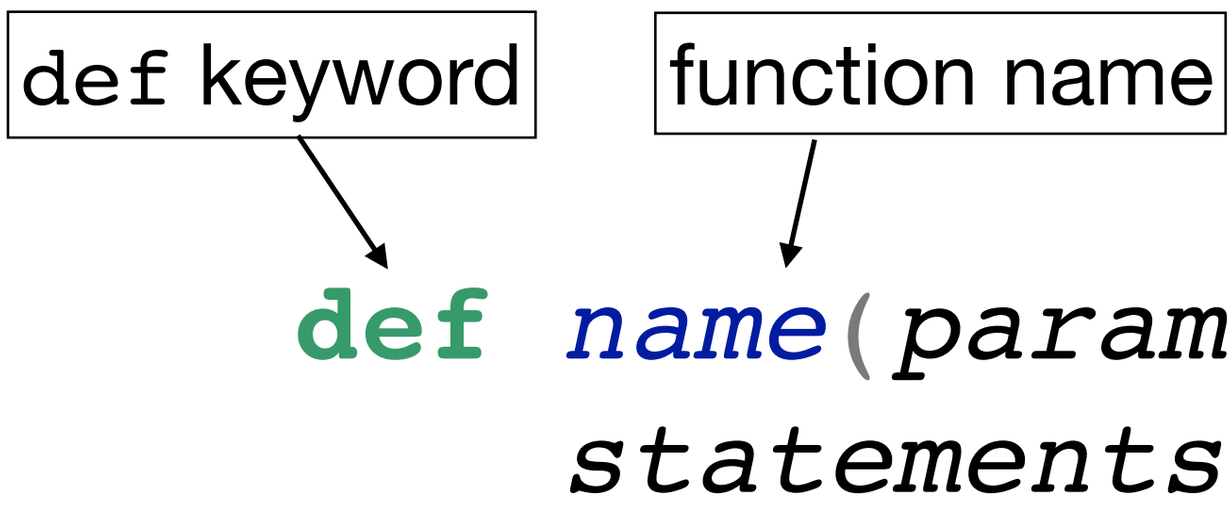
```
def name(parameters):
    statements
```

Two important questions:

1. **How does the function use the arguments (inputs) passed to it?**
2. How does the function return a value?

# Writing Functions: Syntax

**1. How does the function use the arguments (inputs) passed to it?**



# Writing Functions: Syntax

**1. How does the function use the arguments (inputs) passed to it?**

`def` keyword

function name

```
def name(parameters):
    statements
```

**inputs**

comma-separated list of parameters: variable names that will refer to the input arguments

# Why are functions great?

- **Concise** - wrap something complicated in an easy-to-use package:
    - define a function once then easily call it anywhere
- **Customizable** - make the easy-to-use package do different things:
    - customize the task your function performs based on its arguments
- **Composable** - use the result of one computation as input to (or as one step in) another:
    - We'll talk about this next lecture.

# Demo: Function to draw a square using a turtle

# Demo: Function to draw a square using a turtle

- Concise: turtle_square call tells the turtle to do a bunch of things

- Customizable: turtle_rectangle(w, h) function draws a w-by-h rectangle

- add docstrings at the end!

# turtle_rectangle

```
def turtle_rectangle(t, w, h):
    """ Draw a w-by-h rectangle using turtle t
        The turtle starts facing the width
        direction and ends where it started."""
    for i in range(2):
        t.forward(w)
        t.left(90)
        t.forward(h)
        t.left(90)
```

# turtle_rectangle

```
def turtle_rectangle(t, w, h):
    """ Draw a w-by-h rectangle using turtle t
        The turtle starts facing the width
        direction and ends where it started."""
    for i in range(2):
        t.forward(w)
        t.left(90)
        t.forward(h)
        t.left(90)
```

What's `""" this """` about? Two things in one:

# turtle_rectangle

```
def turtle_rectangle(t, w, h):
    """ Draw a w-by-h rectangle using turtle t
        The turtle starts facing the width
        direction and ends where it started."""
    for i in range(2):
        t.forward(w)
        t.left(90)
        t.forward(h)
        t.left(90)
```

What's `""" this """` about? Two things in one:

- **Multiline strings**: An alternate way to write strings that include newlines.

# turtle_rectangle

```
def turtle_rectangle(t, w, h):
    """ Draw a w-by-h rectangle using turtle t
        The turtle starts facing the width
        direction and ends where it started."""
    for i in range(2):
        t.forward(w)
        t.left(90)
        t.forward(h)
        t.left(90)
```

What's `""" this """` about? Two things in one:

- **Multiline strings**: An alternate way to write strings that include newlines.

- A **docstring**: The conventional way to write comments that describe the purpose and behavior of a function.

# Multiline Strings and Docstrings: Demo

# Multiline Strings and Docstrings: Demo

- Multiline strings: printing, assigning, etc.
- A string on a line by itself has no effect on the program.
- Docstrings in functions are like comments (but aren't, technically)

# Docstrings

# Docstrings

Docstrings are **not** required by the language.

# Docstrings

Docstrings are **not** required by the language.

Docstrings **are** required by me.

# Docstrings

Docstrings are **not** required by the language.

Docstrings **are** required by me.

- A docstring tells you **what** the function does, but not **how** it does it.

# Docstrings

Docstrings are **not** required by the language.

Docstrings **are** required by me.

- A docstring tells you **what** the function does, but not **how** it does it.

- In other terms, it tells you what you need to know to **use** the function, but not what the function's author needed to know to **write** it.

# Docstrings: Example

The (actual) source code for turtle.forward:

Docstring:

```
def forward(self, distance):
    """Move the turtle forward by the specified distance.

    Aliases: forward | fd

    Argument:
    distance -- a number (integer or float)

    Move the turtle forward by the specified distance, in the direction
    the turtle is headed.

    Example (for a Turtle instance named turtle):
    >>> turtle.position()
    (0.00, 0.00)
    >>> turtle.forward(25)
    >>> turtle.position()
    (25.00,0.00)
    >>> turtle.forward(-75)
    >>> turtle.position()
    (-50.00,0.00)
    """
```

Implementation:

```
self._go(distance)
```

# Docstrings: Example

The (actual) source code for turtle.forward:

Docstring:

```
def forward(self, distance):
    """Move the turtle forward by the specified distance.

    Aliases: forward | fd

    Argument:
    distance -- a number (integer or float)

    Move the turtle forward by the specified distance, in the direction
    the turtle is headed.

    Example (for a Turtle instance named turtle):
    >>> turtle.position()
    (0.00, 0.00)
    >>> turtle.forward(25)
    >>> turtle.position()
    (25.00,0.00)
    >>> turtle.forward(-75)
    >>> turtle.position()
    (-50.00,0.00)
    """
```

Implementation:

```
self._go(distance)
```

# Docstrings: Example

Python documentation is generated from the docstrings in the code!

`turtle.` **`forward`**`(`*distance*`)`
`turtle.` **`fd`**`(`*distance*`)`

| **Parameters:** | **distance** – a number (integer or float) |
|---|---|

Move the turtle forward by the specified *distance*, in the direction the turtle is headed.

```
>>> turtle.position()
(0.00,0.00)
>>> turtle.forward(25)
>>> turtle.position()
(25.00,0.00)
>>> turtle.forward(-75)
>>> turtle.position()
(-50.00,0.00)
```

# Docstrings: Example

Python documentation is generated from the docstrings in the code!

`turtle.` **`forward`**`(`*distance*`)`
`turtle.` **`fd`**`(`*distance*`)`

| **Parameters:** | **distance** – a number (integer or float) |
|---|---|

Move the turtle forward by the specified *distance*, in the direction the turtle is headed.

```
>>> turtle.position()
(0.00,0.00)
>>> turtle.forward(25)
>>> turtle.position()
(25.00,0.00)
>>> turtle.forward(-75)
>>> turtle.position()
(-50.00,0.00)
```

# QOTD 10/21

```
def pnmr(n, r):
    print(n % r, end=" ")

size = 7
rad = 3
for num in range(0,size):
    pnmr(num, rad)
```

- How many numbers does this print?
- How many 1's does this print?

# QOTD 10/21

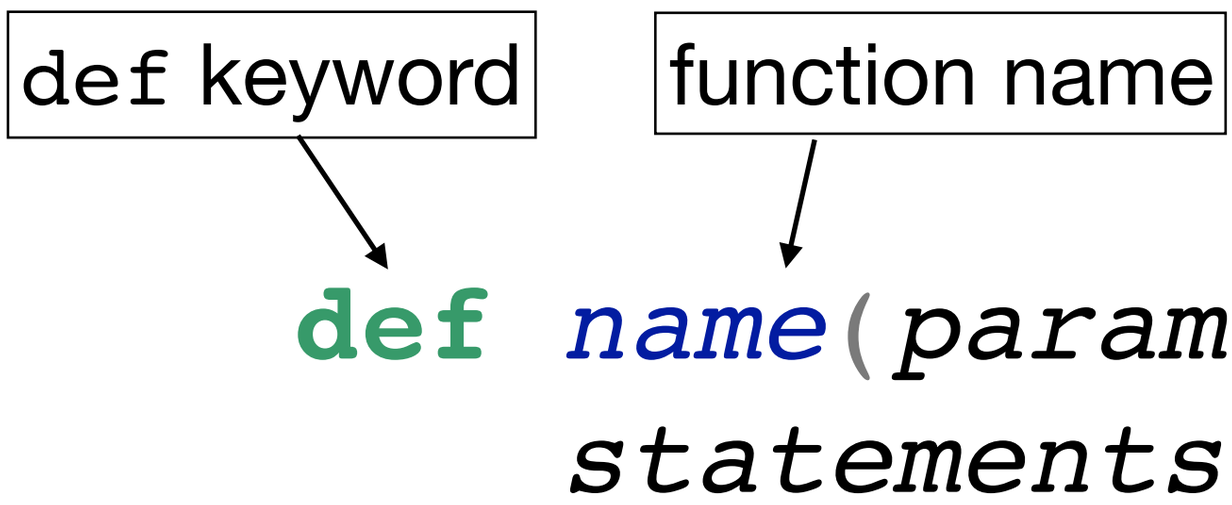
```
def pnmr(n, r):
    print(n % r, end=" ")

size = 7
rad = 3
for num in range(0,size):
    pnmr(num, rad)
```

- How many numbers does this print?
- How many 1's does this print?

Let's step through this using Thonny.

# Writing Functions: Syntax

**1. How does the function use the arguments (inputs) passed to it?**



# Writing Functions: Syntax

**1. How does the function use the arguments (inputs) passed to it?**

`def` keyword

function name

```
def name(parameters):
    statements
```

**inputs**

comma-separated list of parameters: variable names that will refer to the input arguments

# Writing Functions: Syntax

**1. How does the function use the arguments (inputs) passed to it?**

`def` keyword

function name

```
def name(parameters):
    statements
```

**inputs**

comma-separated list of parameters: variable names that will refer to the input arguments

Inside the function, the parameters act as **local variables** that refer to the arguments passed into the function.

# Parameters vs Arguments

Parameters: variable names that will refer to the input arguments.

Parameters (these are new):
variables that take on the value of the arguments

```
def add2(a, b):
    """ Print the sum of a and b """
    print(a + b)


add2(4, 10)
```

Arguments (we've seen these before):
**values** passed into a function.

# Parameters are Local Variables

- They **only** exist inside the function.
- Any other variables declared inside a function are also local variables.
- This is an example of a broader concept called scope: a variable's scope is the set of statements in which it is visible/usable.
- A local variable's scope is limited to the function inside which it's defined.

# Local Variables: Example

**Task:**

Write (define) a function that adds two numbers and prints their sum.

After the function definition, call (invoke) the function.

# Parameters and Local Variables: Demo

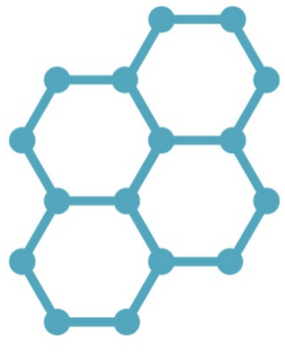
- add2.py

# Parameters and Local Variables: Demo

- add2.py:
  - parameters as local variables (inaccessible outside fn)
  - other local variables
  - variables getting passed in
  - variables shadowing other variables
  - global variables

# Variable Scope

```
1  def print_rectangle_area(width, height):
2      """ Print the area of a width-by-height
3          rectangle """
4
5      area = width * height
6      print(area)
7
8  w = 4
9  h = 3
10 a = w * h
11 print_rectangle_area(w, h)
12
```



# Variable Scope

```
1  def print_rectangle_area(width, height):
2      """ Print the area of a width-by-height
3          rectangle """
4
5      area = width * height
6      print(area)
7
8  w = 4
9  h = 3
10 a = w * h
11 print_rectangle_area(w, h)
12
```

In which line is `area` in scope?

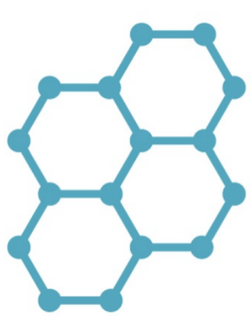
A. 2
B. 6
C. 8
D. 10

# Variable Scope

```
1  def print_rectangle_area(width, height):
2      """ Print the area of a width-by-height
3          rectangle """
4
5      area = width * height
6      print(area)
7
8  w = 4
9  h = 3
10 a = w * h
11 print_rectangle_area(w, h)
12
```

# Variable Scope

```
1  def print_rectangle_area(width, height):
2      """ Print the area of a width-by-height
3          rectangle """
4
5      area = width * height
6      print(area)
7
8  w = 4
9  h = 3
10 a = w * h
11 print_rectangle_area(w, h)
12
```



Which version of line 12 does **not** do the same thing as line 11?

A. `print(h * w)`
B. `print(width * height)`
C. `print(w * h)`
D. `print_rectangle_area(h, w)`

# Variable Scope

# Writing Functions: Syntax

```
def name(parameters):
    statements
```

Two important questions:

1. How does the function use the arguments (inputs) passed to it?
2. **How does the function return a value?**

# Returning values

New statement: the `return` statement

Syntax: `return` *`expression`*

Behavior:

1. *`expression`* is evaluated
2. the function stops executing further statements
3. the value of expression is returned
   i.e., the function call **evaluates** to the returned value

# Function Syntax: Summary

`def` keyword

function name

```
def name(parameters):
    statements
```

**inputs**

comma-separated list of parameters: variable names that will get assigned to the arguments

An indented code block that does any computation, executes any effects, and (optionally) returns a value

**effects; return value**

# Return values: Demo

- Make add2 return the sum instead of printing it.
- Using the result of one computation as the input to another: function composition.

# Returning values

New statement: the `return` statement

Syntax: `return` *`expression`*

Behavior*:*

1. *`expression`* is evaluated
2. **the function stops executing further statements**
3. the value of expression is returned
   i.e., the function call **evaluates** to the returned value

# Returning values

New statement: the `return` statement

Syntax: **`return`** *`expression`*

(can **only** appear inside a function definition)

Behavior*:*

1. *`expression`* is evaluated
2. **the function stops executing further statements**
3. the value of expression is returned
   i.e., the function call **evaluates** to the returned value

# Returning values: Why?

- Next time:

- Using the result of one computation as the input to another: function composition.