Functions

Harry Smith

Learning Objectives

- Be able to read a function's signature to identify its name and its input types
- Be able to follow program execution through multiple function calls
- Be able to write your own functions to perform specific tasks

Functions: a Review

Introduction to Functions

- **Functions** are named lists of statements
- Functions must be **defined** in order to be used. A function definition specifies...
  - the name of the function
  - the arguments that the function takes as input
  - the set of statements that represent what happens when the function is used
- Once defined, a function can be **called** (executed/run)

Many new definitions here,
but about things we already know!

(Re)-Introduction to Functions

We already know about a number of functions and we have some insight into how to use them!

- `print()`
- `len()`
- `range()`
- dozens of PennDraw functions
- plenty more (`sum()`, `max()`, `min()`, `input()`, ...)

(Re)-Introduction to Functions

Typical Usage	Name	Inputs	Returns	Description
`len("lemonworld")`	`len`	a sequence	`int`	Finds the length of a sequence.
`pd.circle(0.5, 0.5, 0.1)`	`circle`	`float`, `float`, `float`	`None`	Draws a circle.
`range(10, 100, 4)`	`range`	`[int]`, `int`, `[int]`	a range	Defines a range with the specified start, stop, and step values.
`print("Hello!")`	`print`	anything at all	`None`	Display a representation of the input(s) as text.

Demystifying Functions

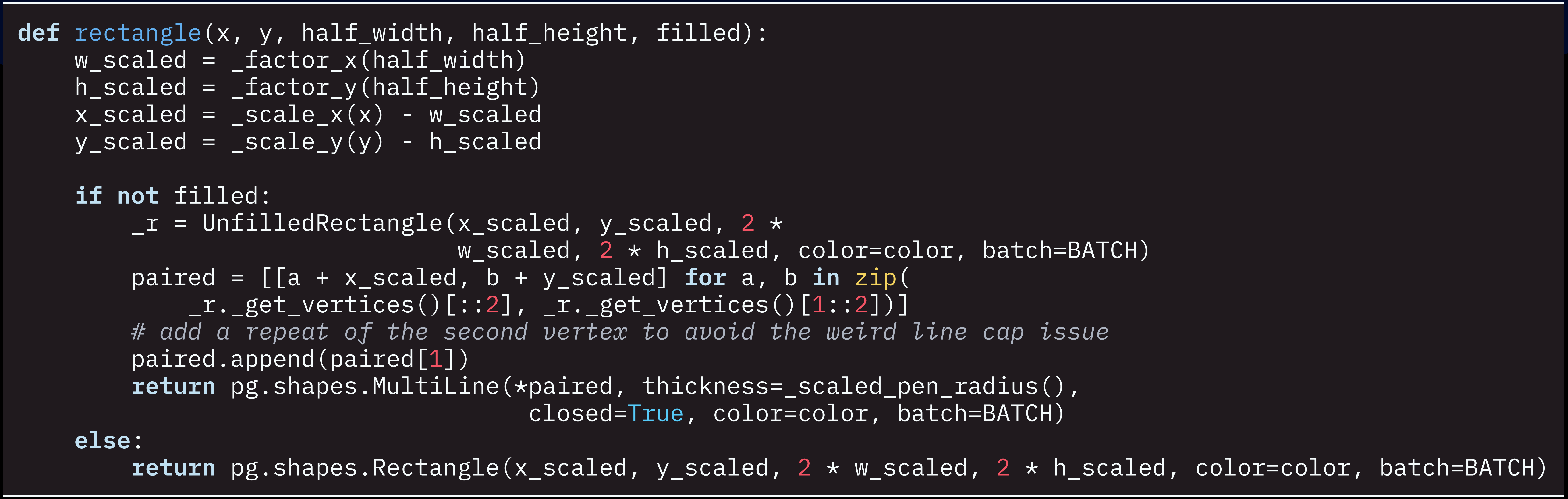
What's happening here?

```
import penndraw as pd
pd.rectangle(0.5, 0.5, 0.1, 0.2)
pd.run()
```

Recall:

- functions are named groups of statements
- those statements are executed when we **call** a function by name

Demystifying (??) Functions



```
def rectangle(x, y, half_width, half_height, filled):
    w_scaled = _factor_x(half_width)
    h_scaled = _factor_y(half_height)
    x_scaled = _scale_x(x) - w_scaled
    y_scaled = _scale_y(y) - h_scaled

    if not filled:
        _r = UnfilledRectangle(x_scaled, y_scaled, 2 *
                               w_scaled, 2 * h_scaled, color=color, batch=BATCH)
        paired = [[a + x_scaled, b + y_scaled] for a, b in zip(
            _r._get_vertices()[::2], _r._get_vertices()[1::2])]
        # add a repeat of the second vertex to avoid the weird line cap issue
        paired.append(paired[1])
        return pg.shapes.MultiLine(*paired, thickness=_scaled_pen_radius(),
                                    closed=True, color=color, batch=BATCH)
    else:
        return pg.shapes.Rectangle(x_scaled, y_scaled, 2 * w_scaled, 2 * h_scaled, color=color, batch=BATCH)
```

Anatomy of a Function

Anatomy of a Function

- **Function definitions** consist of the function's signature as well as a block of statements called its **body**
    - A **function signature** consists of:
        - the function's name
        - the list of parameters that it takes as input.

Dissecting a Function

```python
def multiply_two_numbers(a, b):
    print(f"Multiplying {a} x {b}!")
    product = a * b
    return product
```

The **signature**:

```python
def multiply_two_numbers(a, b):
```

- `def`
- the function's name (`multiply_two_numbers`)
- a pair of parentheses
- a comma-separated list of parameters (`a` and `b`)

Dissecting a Function

```python
def multiply_two_numbers(a, b):
    print(f"Multiplying {a} x {b}!")
    product = a * b
    return product
```

The **body**:

```python
    print(f"Multiplying {a} x {b}!")
    product = a * b
    return product
```

- multiple statements
- all indented one level relative to signature
- uses `a` and `b` as variables without declaring!
- ends with a `return` statement (more on this soon...)

Function Signatures

Function Signatures

```
def <name>(arg0, arg1, ...):
```

- `def`
- function name:
    - chosen to be descriptive of what the function does
    - `snake_case` as always
- pair of parentheses
- comma-separated list of positional parameter names
    - These are the "options" that we specify when calling.
    - Values provided at call available in body using the parameter names specified in the signature.

Function Signatures: Examples

```python
def multiply_two_numbers(a, b):
    ...

def circle(x_center, y_center, radius):
    ...

def say_hello():
    ...
```

Function Signatures Set the Rules for Calling

If this is my signature...

```
def multiply_two_numbers(a, b):
    ...
```

Call	Allowed?
`multiply_two_numbers(4, 5)`	✅
`multiply_two_numbers(4.0, 5)`	✅
`multiply_two_numbers(5)`	❌
`multiply_two_numbers(5, 6, 7)`	❌
`multiply_two_numbers("yes", "no")`	???

Function Signatures Set the Rules for Calling

If this is my signature...

```python
def multiply_two_numbers(a, b):
    ...
```

Call	Allowed?
`multiply_two_numbers(4, 5)`	✓
`multiply_two_numbers(4.0, 5)`	✓
`multiply_two_numbers(5)`	✗
`multiply_two_numbers(5, 6, 7)`	✗
`multiply_two_numbers("yes", "no")`	✓ (but probably will lead to an error down the line...)

Function Signatures Set the Rules for Calling

If this is my signature...

```
def say_hello():
    ...
```

Call	Allowed?
`say_hello()`	✅
literally everything else	❌

Signatures & Calling

If a function signature lists two positional parameters,
it must be called with two positional parameters.

- no restriction on how many parameters a function may require (0 to very many)
- no guarantee about the *types* of the parameters that the function is expecting
    - the joys of Python 🙃

Simple Function Calls

A Worked Example

Here is a function that takes a message and a number and prints that message that number of times.

```python
def print_n_times(msg, n):
    counter = 0
    while counter < n:
        print(msg)
        counter = counter + 1
```

What happens when we call the function: `print_n_times("Hi!", 3)`?

A Worked Example

- The function's *parameters* are `msg` and `n`.
  - These are names for variables that can be used in the body of the function
- The function call provides two **arguments**: `"Hi!"` and `3`
  - These are the values that the parameter variables will take at the start of the function execution.

```
# calling print_n_times("Hi!", 3)
def print_n_times(msg, n):
    # msg = "Hi!"
    # n = 3
    counter = 0
    while counter < n: # while counter < 3:
        print(msg)     # print("Hi!")
        counter = counter + 1
```

Function Calls & Arguments

When a function is called, the values of the arguments provided with the call are associated *in order* with the parameters in the function definition

- this gives the parameter variables their initial values in the function body
- allows each individual call to change the behavior of your output
  - `print_n_times("Hi!", 3)` prints `"Hi!"` three times
  - `print_n_times("Bye!", 2)` prints `"Bye!"` two times

return

`return`

Function calls are themselves *expressions*, meaning that they always have a value.

- The value of a function call is determined by the value that function **returns**

`return` is keyword that serves two purposes:

- stops function execution in its tracks
- provides a value for the expression of the function call

return: An Example

```
def multiply_two_numbers(a, b):
    print(f"Multiplying {a} x {b}!")
    product = a * b
    return product
```

If we write the call `multiply_two_numbers(3, 7)`, then...

```
    # a = 3
    # b = 7
    print(f"Multiplying {a} x {b}!")
    product = a * b                     # product = 3 * 7
    return product                      # return 21
```

...we return the value of `product`, which is `21` based on this function call. The following therefore evaluates to `True`:

```
multiply_two_numbers(3, 7) == 21
```

Printing vs. Returning

An output that's *printed* is not the same as an output that's *returned.*

- Any call to `print()` will make text appear on the screen, but it doesn't produce a value
- If a function is supposed to calculate and create some value (e.g. the product of two numbers), it must *return* that value in the function body.