Objectives

- Development Processes with Functions
  - Bottom-up
  - Refactoring
- Broader Issue: Algorithmic Bias

Review

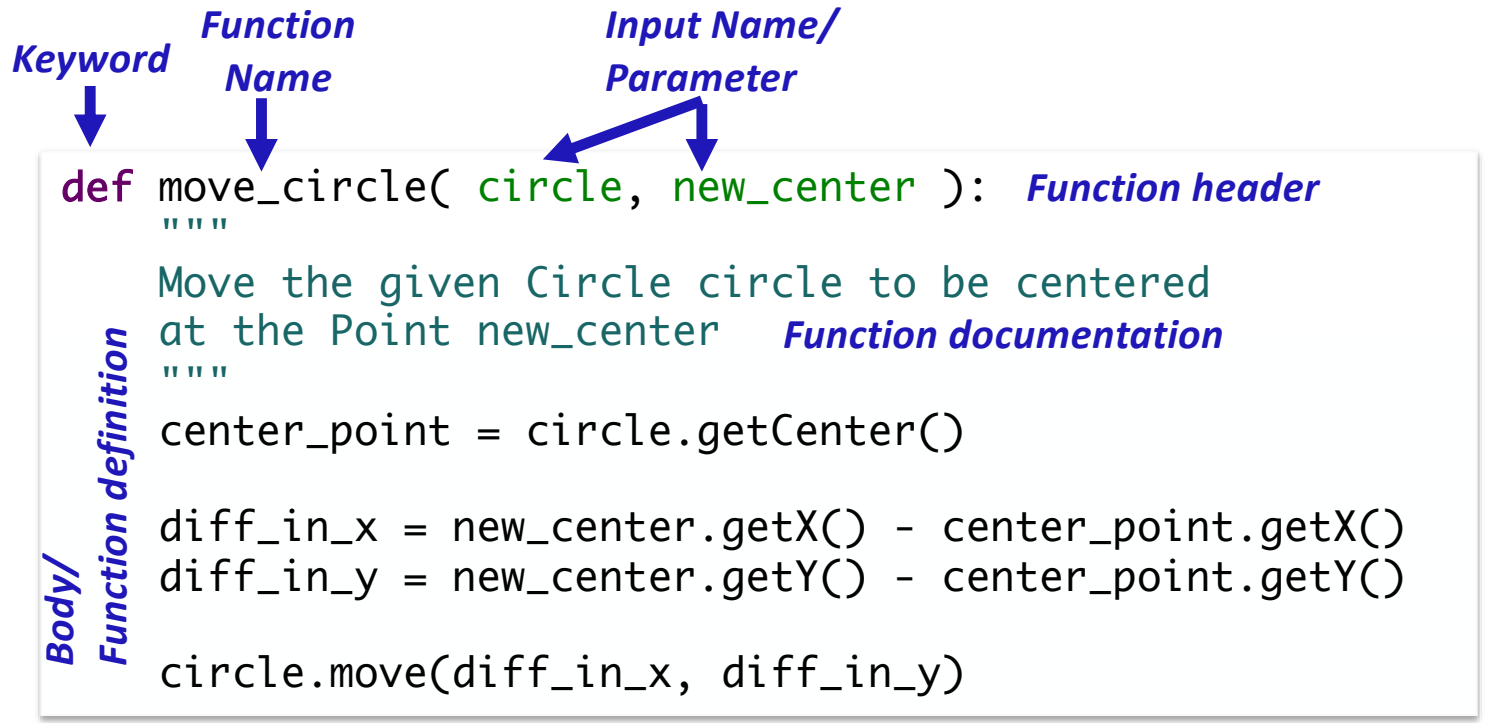
1. What are benefits of functions?
2. What is the syntax for creating our own functions?
   - How do we indicate that our function requires input?
   - How do we indicate that our function has output?
3. How is output from a function different from output from a program?
4. What does a variable's "*scope*" mean?
5. What are characteristics of a good function?
6. What should the content of a docstring be?
7. How can we programmatically test our functions?

Review: Why Write Functions?

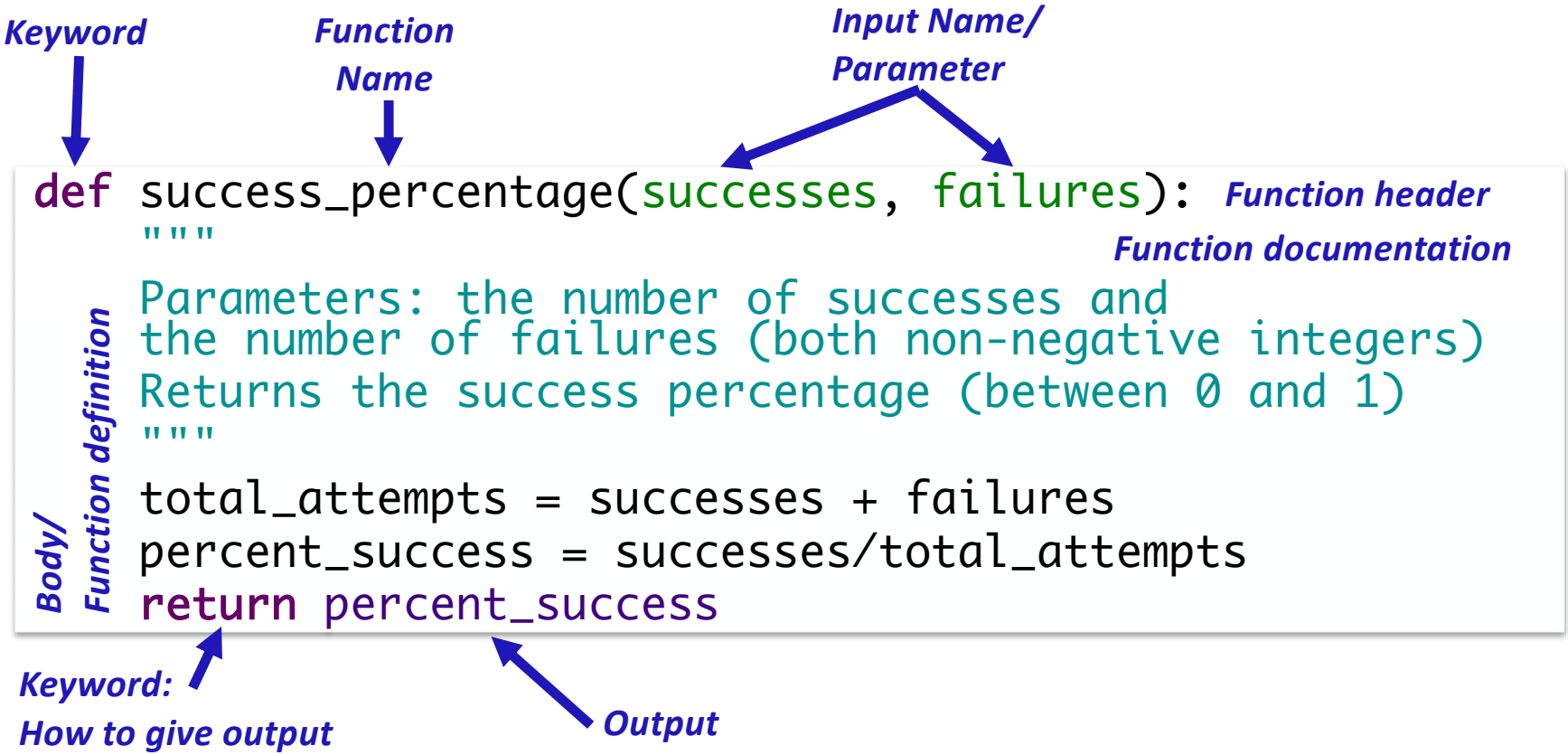
Functions do not allow you to solve any new problems, so why write them?

- Allows you to break up a problem into *smaller*, more *manageable* parts
- Makes your code easier to *understand*
- Hides implementation details (*abstraction*)
  - Provides *interface* (input, output)
- Makes part of the code *reusable* so that you:
  - Only have to write function code once
  - Can debug it all at once
    - Isolates errors
  - Can make changes in one function (*maintainability*)

Function Definition Example Without Output



Function Definition Example With Output



Review: `return` vs `print`

- In general, whenever we want output from a function, we'll use `return`
  - Results in a more flexible, reusable function
  - Let whoever called the function figure out what to display
- Use `print` for
  - Debugging your function (then remove)
    - Otherwise, unintended side effect of calling the function
  - When you have a function that is supposed to display something
    - Sometimes, that is what you want.

Review: Writing a "Good" Function

- Should be an "intuitive chunk"
  - Doesn't do too much or too little
  - If does too much, try to break into more functions
- Should be reusable
- Should have a descriptive, "action" name
- Should have a comment that tells what the function does

Review: Writing Documentation for Functions

- Tell the function caller how to use the function
- Include the function's pre- and post- conditions
  - **Precondition**: Things that must be true for function to work correctly
    - E.g., num must be even; circle must be a Circle object
  - **Postcondition**: Things that will be true when function finishes if the precondition is true
    - E.g., the returned value is the max; circle will be moved to the new point
- Again, the exact format matters less than the content

Review: Example Documentation

- Describes at high-level
- Describes parameters

Note: does not discuss implementation

```
def printVerse(animal, sound):
    """
    Prints a verse of Old MacDonald, plugging in the
    animal and sound parameters (which are strings),
    as appropriate.
    """
    print(BEGIN_END + EIEIO)
    print("And on that farm he had a", animal, EIEIO)
    …
```

Comment style: **Docstring**
"documentation string"

When you call the `help` function, it shows the docstrings.

Review: Another Example Docstring

- Describes at high-level
- Describes parameters

Note: does not discuss implementation

```
def success_percentage(successes, failures):
    """
    Parameters: the number of successes and
    the number of failures (both non-negative integers)
    Returns the success percentage (between 0 and 1)
    """
    total_attempts = successes + failures
    percent_success = successes/total_attempts
    return percent_success
```

Comment style: **Docstring**
"documentation string"

When you call the `help` function, it shows the docstrings.

Review: Write Docstring for sumEvens

```
def main() :
   x=10
   sum = sumEvens( x )
   print("The sum of even #s up to", x, "is", sum)

def sumEvens(limit) :
   """
   Returns the sum of even numbers from 0 up to but
   not including limit, which is an integer
   """
   total = 0
   for x in range(0, limit, 2):
         total += x
   return total

main()
```

Many other correct doc strings

Review `test` Module

- Not a standard module
  - Included with our textbook
  - More sophisticated testing modules exist but works well for us
- Function:
  - `testEqual(actual, expected[, places=5])`
    - Parameters: actual and expected results for a function.
      - `places`: the number of decimal places to match when comparing floats
    - Displays "Pass" and returns True if the test case passes.
    - Displays error message, with expected and actual results, and returns False if test case fails.

Review: Testing sumEvens

```
import test
…
def testSumEvens():
    actual = sumEvens( 10 )
    expected = 20
    test.testEqual( actual, expected )
    test.testEqual( sumEvens(12), 30)

def sumEvens(limit):
    total = 0
    for x in range(0, limit, 2):
        total += x
    return total
```

This is the actual result from our function

This is what we expect the result to be

What are other good test cases?

testSumEvens.py

Another Example of Programmatic Testing

- Testing a constructor/function/method that affects state:

```
def testGraphWin():
    window = GraphWin("Title", 300, 200)
    test.testEqual( window.getWidth(), 300 )
    test.testEquals( window.getHeight(), 200 )
    …
```

  - Call the constructor/function/method under test
  - Check the resulting state

EVOLVING DEVELOPMENT PROCESSES

Evolving General Design Patterns

- Original general design pattern:
  1. Optionally, get user input
  2. Do some computation
  3. Display results
- Alternative general design pattern:
  1. Optionally, get user input
  2. Do some computation by calling **functions**, get results
  3. Display results

Development Process: Bottom-Up

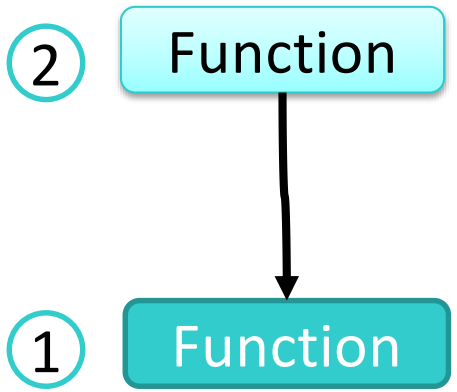
1. Define a function
   - Document
   - Test the function

Focus on just a part of the larger problem

Development Process: Bottom-Up

2. Use the function in context/ call the function

1. Define a function
   - Document
   - Test the function



Practice

1. Consider a function to calculate our favorite expression: $i^2 + 3j - 5$
   a. What does the function do?
   b. What is its input?
   c. What is its output?
   d. Write its documentation
2. Write test function for function
3. Implement the function
   - Update documentation, if needed
4. Test the function
   - Debug, retest
5. Use the function

`our_favorite_expression.py`

Another Bottom-Up Development Example

1. Define, document, and test a function that
   - Given two numbers
   - Returns the average of those two numbers

2. Create a program that
   - Prompts for those two numbers
   - Displays the the average

`average2.py`

REFACTORING

Refactoring

- After you've written some code and it passes all your test cases, the code is probably still not perfect
- ***Refactoring*** is the process of improving your code *without* changing its functionality
  - Organization
  - Abstraction
    - Example: Easier to read, change
  - Easier to test
- Part of iterative design/development process

Refactoring into Functions

- Symptom: you note that there is some functionality that would benefit from being a function
- Motivation: improve readability and reusability of your programs
- Where to refactor with functions
  - Duplicated code
    - "Code smell"
  - Reusable code
  - Multiple lines of code for one purpose

Refactoring:
Converting Functionality into Functions

1. Identify functionality that should be put into a function
   - What is the function's input?
   - What is the function's output?
2. Define the function
   - Write comments
3. Call the function where appropriate
4. Create a `main` function that contains the "driver" for your program
   - Put at top of program
5. Call `main` at bottom of program

Example: PB & J

1. Gather materials (bread, PB, J, knives, plate)
2. Open bread
3. Put 2 pieces of bread on plate
4. Spread PB on one side of one slice
5. Spread Jelly on one side of other slice
6. Place PB-side facedown on Jelly-side of bread
7. Close bread
8. Clean knife
9. Put away materials

- Which of these are the "core" part of making a PB & J sandwich?
- How would you describe the rest of the parts?

Example: PB & J

1. Gather materials (bread, PB, J, knives, plate)
2. Open bread
3. Put 2 pieces of bread on plate
4. Spread PB on one side of one slice
5. Spread Jelly on one side of other slice
6. Place PB-side facedown on Jelly-side of bread
7. Close bread
8. Clean knife
9. Put away materials

Example: PB & J as Functions

1. Gather materials (bread, PB, J, knives, plate)
2. Open bread
3. Put 2 pieces of bread on plate
4. Spread PB on one side of one slice
5. Spread Jelly on one side of other slice
6. Place PB-side facedown on Jelly-side of bread
7. Close bread
8. Clean knife
9. Put away materials

```
def main():
    prepare()
    pbj = make_pbj_sandwich()
    clean_up_supplies()
main()
```

Example: PB & J as Functions, 10 x

1. Gather materials (bread, PB, J, knives, plate)
2. Open bread
3. Put 2 pieces of b
4. Spread PB on on
5. Spread Jelly on o
6. Place PB-side fac
7. Close bread
8. Clean knife
9. Put away materials

```
def main():
    prepare()
    for sandwich in range(10):
        pbj=make_pbj_sandwich()
    clean_up_supplies()
main()
```

Refactoring in Practice

Original file with code to be refactored

New (empty) program file

1. Copy relevant code from original file into the new file
2. Convert that code into a function in the new file
3. Test the function programmatically
4. Copy other code from original file into new file, replacing the functionality with a call to the newly defined function

Refactoring Practice

- Refactor the code from your lab with our favorite expression to create a function to calculate the expression and to have a main function

  - We should get a similar end result to what we had with the bottom-up approach

Refactoring: An Iterative Process

- As you refactor, you'll often note new places to refactor
- Example: after extracting functionality into a function, you'll realize that it would be helpful to put the rest of your code in a `main` function

Summary: Development Approaches

- There are several development approaches
- Not mutually exclusive
- Often will switch between them, depending on circumstances
- As programs grow in size, there is no “one way” to write code
  - But there may be better ways to make progress
  - If you’re stuck, step back and reassess your approach

Debugging Advice

- Build up your program in steps
  - Always write only small pieces of code
  - Test, debug. **Repeat**
- Write function body as part of `main`, test
  - Then, separate out into its own function
  - Similar to process using in lab probs
- Test function separately from other code

BROADER ISSUE: ALGORITHMIC BIAS

Broader Issue Groups

Pod 1	Pod 2	Pod 3	Pod 4	Pod 5
Ilaria Jaz Rowen Ruoan	Berkley Brielle Julia Juyoung	Ben Rosen Hudson Sam Wesley	Abrar Ben Teague Brett Caleb	Cheng Devin Liliane Renee

https://www.youtube.com/watch?v=1zXFlyuuCDI

Algorithmic Bias

- Comment on the following statement in context of CSCI111: “Algorithms are opinions embedded in code.”
- Reflect on “My department of education contact told me, ‘It’s math and I wouldn’t understand it.’”
  - Why is it beneficial to make the algorithm transparent? To keep it opaque?
- For the sentencing algorithm that considered likelihood of recidivism
  - What are the tradeoffs of using an algorithm vs the judge making a decision? (the judge ultimately does make the decision)
  - What should be considered in sentencing?
  - How do we/should we “interrogate” algorithms?
- What algorithm are you questioning now?

Algorithmic Bias

- You're learning more about programming and algorithms
- It's a good idea to stop and question algorithms and results
- You'll be a purchaser of software
- I want you to be informed and ask good questions when making decisions
  - Yet another benefit of the liberal arts!

Exam Friday

- **Do not panic**
- In-class, on paper
  - Emphasis on critical thinking
  - Lab was to experiment and cement you're learning. Now you're ready!
- Exam Preparation Document is on course web page
- Similar problems to class and lab
  - Review questions
  - Worksheets
  - Problems
- Content: up through Tuesday's lab
- No broader issue this week

Looking Ahead

- Pre-Lab due before lab on Tuesday
- Exam Friday