Welcome!

We'll get started shortly. Please take the Zoom poll in the meanwhile!

CS 49 Week 7

Surajit A. Bose

Agenda

- Functions:
    - Function input: parameters, arguments
    - Function output: **`return`** statements and values
    - Functions as black boxes
- Worked example: In_Range
- Revised coding template
- Section problem: Medical Test Simulator

How to get hold of me / get help from other resources

- Surajit's office hours
  - Fridays 12 noon–1p, directly after section
  - By appointment on Zoom
- Pronto DM for a quick response (usually within a couple hours)
- Email bosesurajit@fhda.edu or Canvas inbox, 24 hr turnaround
- The github repo has section materials, starter code, and solutions

---

- Pronto DM or Canvas inbox for Lane
- Lane's office hours
- Online or in-person tutoring at the STEM center (Room 4213)

Review: Decomposition and Problem Solving

The process of problem solving: stepwise refinement

- First, **understand** the pre- and postconditions
- While **designing the solution**, identify the repeatable, self-contained building blocks that will be the functions: **decompose** thoroughly
- When coding, assume the functions are done
    - Write out the function headers
    - Use the `pass` keyword for the bodies
- Assemble the functions in `main()` to solve the overall problem
- **Implement** each building block by writing out its function body
- **Test** iteratively and **refine** the solution:
    - Check that each function works as expected
    - Check that all functions put together in `main()` solve the entire problem as expected

Functions in Python

Function input and output (Slide 1 of 2)

- Suppose that in a program, we have to repeatedly check pairs of numbers to see which number is smaller
- Since this action is repeated many times in our program, we want to write a function for this
- Every time **`main()`** has two numbers to compare, it calls **`smaller()`** to do the actual comparison. **`smaller()`** then tells **`main()`** which of the two numbers is smaller
- Problems:
    - Input: how does **`main()`** get the pair of numbers to **`smaller()`**?
    - Output: how does **`smaller()`** get the smaller number back to **`main()`**?

Function input and output (Slide 2 of 2)

- To get data in: a function takes in zero or more **parameters** as input
- To get data out: a function **returns** zero or one values as output
- E.g., here is a function for returning the smaller of two numbers:

```
def smaller(num1, num2):
    if num1 < num2:
        return num1
    return num2
```

- `num1` and `num2` are parameters or the expected input to the function
- The smaller number of the two is the `return` value or the output from the function

Function structure

- The function header has the keyword **`def`**, the function name, and parentheses with the parameter list, followed by a colon :

```
def smaller(num1, num2):
```

- Indented below the header is the function body, which does the required work using the input values:

```
    if num1 < num2:
        return num1
    return num2
```

- The function has zero or more return statements depending on its work. E.g., this one could be designed to just print the smaller number, not return it, in which case we would **`print()`** and not **`return`**

Function calls (Slide 1 of 2)

- A function is **called** from within another function, the **calling function** or **caller**
- When calling the called function, the caller passes in appropriate **arguments** for the expected parameters, then handles the return value if there is one
- Here, **`main()`** is the caller and **`smaller_num()`** the called function:

```
def main():
    first_num = int(input('Enter first number: '))
    second_num = int(input('Enter second number: '))
    smaller_num = smaller(first_num, second_num)
    print(f'The smaller number is {smaller_num}')
```

- **`first_num`** and **`second_num`** are arguments passed from **`main()`** to **`smaller()`**
- The value returned from **`smaller()`** to **`main()`** is stored and then printed out

Functions as black boxes

- The function is a black box; the caller does not know how the called function does its work.
- E.g., **`smaller()`** could be defined in these alternative ways (among others):

```python
def smaller(num1, num2):
    if num1 > num2:
        return num2
    return num1
```

```python
def smaller(num1, num2):
    if num1 <= num2:
        return num1
    return num2
```

- From the standpoint of the calling function, it does not matter what happens inside the called function as long as the result is as expected

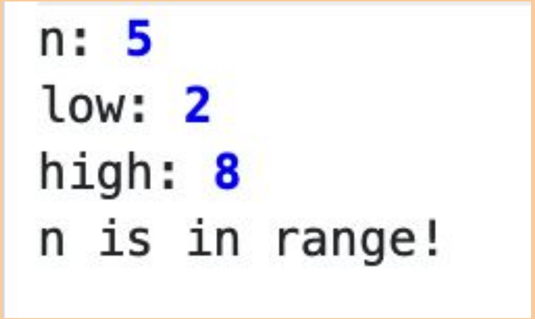
Advantages of using functions

- The three Rs: readability, repeatability, reliability
- Compare these two programs that do exactly the same thing
  - **`logical_operators.py`**
  - **`logical_operators_func.py`**
- The program that uses a separate function to evaluate and print the passed in arguments is much shorter, clearer, and cleaner than the program that does everything in **`main()`**
- There is also less risk of error in the program that uses a separate function
- A template for writing code that uses functions with parameters and return values is at the github repo

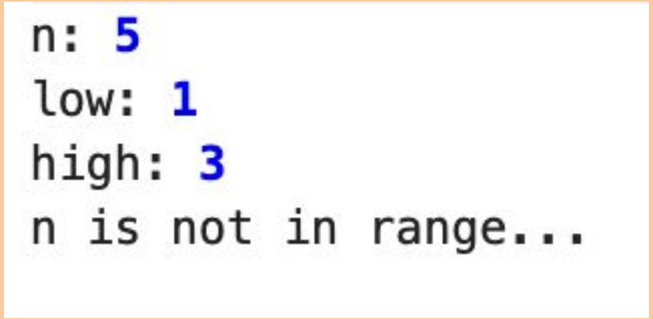
Worked Example: In range

In range (Slide 1 of 2)

- In **`main()`**, prompt the user for three numbers, **`n`**, **`low`**, and **`high`**
- Call **`in_range()`** to check whether **`n`** is in the range between **`low`** and **`high`**, inclusive
- Back in **`main()`**, print whether **`n`** is in range or not
- User input is in **`blue`**:



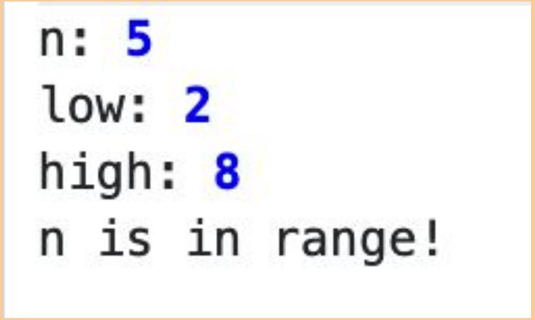
```
n: 5
low: 2
high: 8
n is in range!
```



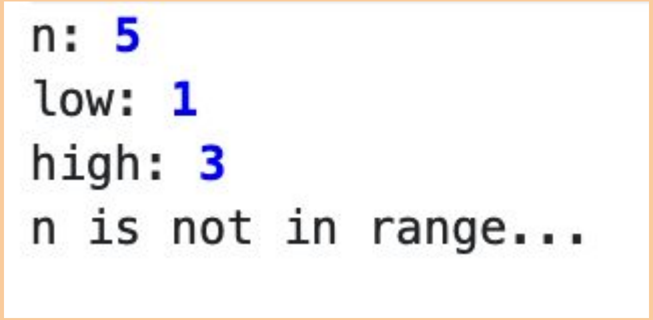
```
n: 5
low: 1
high: 3
n is not in range...
```

In range (Slide 2 of 2)

- In **`main()`**, to what type should the user inputs be cast?
- How many parameters should **`in_range()`** expect? Of what type?
- What should **`in_range()`** return? What type is this return value?
- How should **`main()`** use this return value to determine the screen output?
- Starter code for **`in_range()`** is at the github repo



```
n: 5
low: 2
high: 8
n is in range!
```



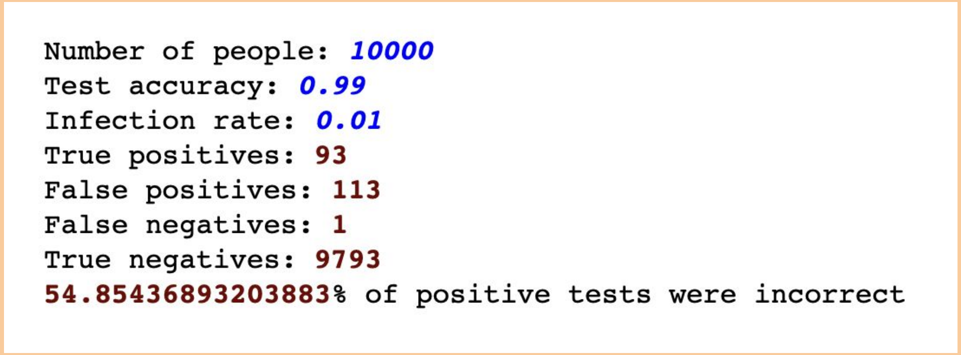
```
n: 5
low: 1
high: 3
n is not in range...
```

Section Problem: Medical Test Simulator

Medical Test Simulator (Slide 1 of 4)

- Given:
  - A population of 10,000 individuals
  - An infection rate of 1% for a disease
  - A diagnostic test with 99% accuracy

… what percentage of positive results from the test will be false positives?



```
Number of people: 10000
Test accuracy: 0.99
Infection rate: 0.01
True positives: 93
False positives: 113
False negatives: 1
True negatives: 9793
54.85436893203883% of positive tests were incorrect
```

Notes:

- User input is in **`blue`**
- With probabilistic simulations, results will be different each time the test is run

Medical Test Simulator (Slide 2 of 4)

- Starter code is at the github repo
- In **`main()`**, prompt the user for these three numbers:
  - number of people
  - test accuracy
  - infection rate
- In **`simulate_tests()`**:
  - Use the given probabilities and simulate a diagnostic test on each individual
    - Randomly determine whether the individual is infected
    - Randomly determine whether the test is accurate

Digression: Simulating a probabilistic outcome

- The assignment provides this hint: the expression **`random.random() < prob`** evaluates to **`True`** with probability **`prob`**
- Explanation:
    - The range 0 through 9 comprises 10 integers
    - Of those, 8 integers (0 through 7 inclusive) are below 8
    - So if we run **`random.randint(0, 9)`**, there is an 8 in 10 chance that the integer we get will be below 8. Eight in 10 is 8/10 or 80%.
    - Likewise, **`random.random()`** returns a float from 0 up to but not including 1
    - Therefore, given a probability of 0.8, 80% of the time the random float will be below 0.8

Medical Test Simulator (Slide 3 of 4)

- In **`simulate_tests()`**, contd.:
  - Calculate and print these four values:
    - true positives
    - false positives
    - false negatives
    - true negatives
  - Return the proportion of incorrect positive results
- Back in **`main()`**:
  - Print out the percentage of incorrect positive tests

Medical Test Simulator (Slide 4 of 4)

- What type(s) should user input be cast to in **`main()`**?
- How many parameters should **`simulate_tests()`** expect? Of what type(s)?
- How should we calculate the four necessary values of true and false positives and negatives?
- Do we need a loop? If so, for what?
- What is the formula for the **`return`** value, i.e., the proportion of incorrect positive results?
- How can we display this as a percentage in **`main()`**?

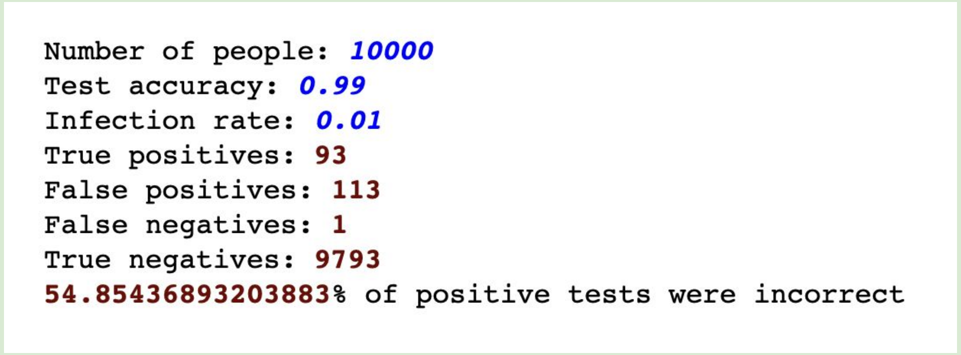
Bonus Slides

Medical Test Simulator and Data Science

Confusion matrices

- In data science, this sort of tally is called a **`confusion matrix`**

	Diagnosed as infected	Diagnosed as not infected
Actually infected	93 (true positives)	1 (false negative)
Actually not infected	113 (false positives)	9793 (true negatives)



```
Number of people: 10000
Test accuracy: 0.99
Infection rate: 0.01
True positives: 93
False positives: 113
False negatives: 1
True negatives: 9793
54.85436893203883% of positive tests were incorrect
```

This test has good **`recall`** or **`sensitivity`**: 93 out of 94 infected individuals were identified (98.94%).

This test has poor **`precision`**: 93 out of 206 individuals identified as infected were infected (45.15%)

This test has good **`specificity`**: 9793 out of the 9906 uninfected individuals were correctly identified (98.86%)

The base rate fallacy

- The assignment asks: "As you can see, the results are pretty surprising: nearly 55% of people told they had the disease didn't actually have it. Intuitively, why might this be? Is our result just anomalous?"
- This is the accuracy paradox, aka the false positive paradox or base rate fallacy
- Given the infection rate of 1% in a population of 10,000:
  - 100 people are infected
  - 9,900 are uninfected
- Given the test accuracy rate of 99%:
  - Of the 100 infected people, we will likely have 99 true positives and 1 false negative
  - Of the 9,900 infected people, we will likely have 9,801 true negatives and 99 false positives
  - Of the total 198 positive results, likely half or 50% of them will therefore be false positives!
- Reminder: with probabilistic simulations, results will be different each time and the results won't be exactly as mathematically predicted