#4: Nesting, Testing, and Debugging

SAMS SENIOR CS TRACK

Last Time

Used conditionals to change program actions based on data

Used loops to repeat actions within a program

Today's Learning Goals

Combine blocks of code to create complex programs

Use testing to find where our programs don't work

Use debugging to fix programs when they're broken

Nesting

Combining Blocks of Code

We now know how to write code using conditionals and loops. However, we haven't yet tapped into their true potential- **combining** the blocks of code to create complex actions.

We've already seen a bit of this potential by putting conditionals and loops in functions. However, we can also nest conditionals and loops **within each other**.

Indentation

In Python, indentation is used to show which block a statement belongs to. Functions, conditionals, and loops all have bodies which can contain variable numbers of statements, which is why we call them blocks.

Example: how many lines are in the while loop? the for loop? the if statement?

```
while x != 0:
    print("boo")
    x = x // 2
for i in range(10):
    if x % i == 0:
        x += 1
    x = x * 2
x = x + 1
```

Nested Conditionals

When we nest conditionals, both conditionals need to be True to reach the innermost statement.

Example: write a program that returns True if the age and driving status of the person allows them to drink. How many different ways can we arrange these statements?

```
def canDrinkAlcohol(age, isDriver):
    if age >= 21:
        if isDriver == False:
            return True
        else:
            return False
    else:
        return False
```

Nested Loops

When we nest loops, we repeat the inner loop every time the outer loop takes a step.

Example: print all the coordinates on a plane from (0,0) to (5,5)

```
for x in range(5):
    for y in range(5):
        print("(", x, ",", y, ")")
```

Note that every iteration of y happens anew in each iteration of x.

Nested Functions

We don't 'nest' functions the way we nest conditionals and loops, but we do write helper functions to reduce the size of our functions in code!

Example: write the function nthPrime(n), which finds the nth prime number. nthPrime(1) is 2, nthPrime(3) is 5, etc.

Instead of writing all the code in one function, we can write isPrime(n), and use it to help solve nthPrime(n)!

Example: nthPrime

```
def nthPrime(n):
    guess = 0
    found = 0
    while found < n:
        guess += 1
        if isPrime(guess):
            found += 1
    return guess
```

```
def isPrime(n):
    if n < 2:
        return False
    for factor in range(2, n):
        if n % factor == 0:
            return False
    return True
```

Example: Conditionals in Loops

We can combine different block statements inside each other as well!

Example: return only the lowercase letters in the inputted string

```
def onlyLower(s):
    t = ""
    for c in s:
        if "a" <= c <= "z":
            t += c
    return t
```

Testing

Finding Errors

The only way to find out whether your code has an error is to **test** it. Testing is a large part of the programming process!

Simple tests check whether a function returns the expected **output** when given a specific **input**. We can write these using **assert** statements. If the statement is True, nothing happens; if it is False, the statement crashes.

Example:

```
assert isPrime(3) == True
```

Testing

When writing test functions, we need to cover **likely cases where things can go wrong**. If we don't, our program might develop a bug without us realizing!

In particular, you should always try to cover:

- **Normal cases** – provided and obvious examples
- **Large cases** – larger-than-usual input
- **Edge cases** – pairs of input that result in opposite choices in the code
- **Special cases** – 0 and 1, empty string, unexpected types
- **Varying results** – make sure that all your test cases don't return the same result!

Testing isPrime

Think/Pair/Share: Let's come up with test cases for each of these categories for isPrime.

Normal case:

Large case:

Edge case:

Special case:

Varying results:

Test first!

There's a temptation when programming to write the code first, then test it when you're done.

It's actually much more useful to write the tests first, then write the code! Writing the tests will help you better understand what the code needs to do.

This is called **test-driven development**.

Debugging

Debugging

Debugging

Debugging is the process of determining **where** your code is not working correctly, figuring out **why** it is incorrect, and **fixing** the error.

In general, while debugging, the best thing you can do is **read the error messages and code carefully**. However, different errors are best fixed with different approaches.

Remember the three types of errors from last week: **syntax errors**, **runtime errors**, and **logical errors**.

Debugging Syntax Errors

When your program encounters a **syntax error**, follow the following steps:

1. Read the error message and verify that this is a SyntaxError.
2. Look for the line number and the arrow pointing at the code to find the error's location.
3. Carefully read the line of code to find the incorrect syntax.

Example: Syntax Debugging

```
1: x = 5
2: if x > 0
3:     print("Positive")
4: else
5:     print("Negative")
```

```
Running script: "/Users/krivers/samstmp.py"
  File "/Users/krivers/samstmp.py", line 2
    if x > 0
         ^
SyntaxError: invalid syntax
```

Debugging Runtime Errors

When your program encounters a **runtime error**, follow the following steps:

1. Read the error message and identify the type of error.
2. Look for the line number, go to that line of code, and identify which part might be associated with the error.
3. Use print statements to identify what the code's **state** is at that point in the program, and work out what the state should actually be.
4. Identify how to change the program to achieve the desired state.

Example: Runtime Debugging

```
1: friend = "Stella"
2: for letter in freind:
3:     print(letter + "!")
```

```
Running script: "/Users/krivers/samstmp.py"
Traceback (most recent call last):
  File "/Users/krivers/samstmp.py", line 2, in <module>
    for letter in freind:
NameError: name 'freind' is not defined
```

Debugging Logical Errors

When your program encounters a **logical error**, follow the following steps:

1. Don't start with the error message, it won't be helpful. Instead, **identify the input, expected output, and actual output of the failing test case**.
2. Make sure you understand **why** the program should achieve the expected output on the given input.
3. Add **print statements** to your code at important junctures to visualize the program's state as it runs.
4. Compare the printed state to the expected state, find where the two diverge, and use problem solving to determine how your algorithm needs to be changed to fix the state.

Example: Logical Debugging

```
1: def containsUpper(s):
2:     for c in s:
3:         if "A" <= c <= "Z":
4:             return True
5:         else:
6:             return False
7:
8: s = "hello Mr. Bond"
9: assert(containsUpper(s) == True)
```

```
Running script: "/Users/krivers/samstmp.py"
Traceback (most recent call last):
  File "/Users/krivers/samstmp.py", line 9, in <module>
    assert(containsUpper(s) == True)
AssertionError
```

Today's Learning Goals

Combine blocks of code to create complex programs

Use testing to find where our programs don't work

Use debugging to fix programs when they're broken