#3: Algorithmic Thinking

SAMS PROGRAMMING C

Review from Last Week

Understand how to write data, operations, variables, functions, conditionals, and loops in Python

Use these constructs to build simple programs

Today's Learning Goals

Combine blocks of code to create complex programs

Use the problem-solving process to develop solutions to new problems

Use testing and debugging to verify that our programs work

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**.

Example: Nested Conditionals

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
```

Example: Nested Loops

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.

Example: Loops in Conditionals

Example: if the given number is postive, print the numbers up to that number

```
def printToN(n):
    if n > 0:
        for i in range(n):
            print(i)
```

Example: Conditionals in Loops

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
```

Problem Solving

Programming in general involves determining **how to solve problems with algorithms**. An algorithm is a defined process that accomplishes some task. When we write Python code, we are encoding the process into a language the computer can understand.

Doing problem solving in programming can be broken down into the following steps:

1. Understand the problem
2. Devise a plan
3. Carry out the plan
4. Review your work

Understanding the Problem

The first step of problem solving is to **thoroughly undestand the problem**. This sounds simple, but can be tricky. If you thoroughly understand a problem, you should understand what the function will do on *any given input*. This will involve carefully reading the WHOLE prompt.

Example: "Write the function countLowercaseUpToPercent(s) that takes a possibly-empty string and returns the number of lowercase letters that occur in the string before the first percent sign (%)." What should countLowercaseUpToPercent(s) return when given the value...

- "abc%def"
- "ABC%DEF"
- "abcDEF%"
- "12%34"
- "abcd"
- ""
- 4

Devising a Plan

Once you understand a problem, you need to devise an algorithm or plan for how to solve it. Depending on the prompt, you may need to **translate a provided algorithm**, **apply a known algorithm pattern**, or **generate a new algorithm**.

Translate: the prompt will describe an algorithm in regular language, and you will only need to translate it into Python. For example, "write a function that computes the distance between two points". You don't need to invent a distance algorithm- one already exists!

Apply: the prompt may be similar to a problem you've seen before; for example, "return only the uppercase characters in the given string". Then you can apply the previous code pattern and modify it as necessary.

Generating New Algorithms

The hardest problems are those in which the algorithm to solve the problem is not immediately familiar. In these cases, you will need to invent an algorithm yourself. The best way to learn how to generate algorithms is to **practice**, as each problem has its own individual quirks. However, there are two helpful approaches you can use when generating algorithms: **induction** and **top-down design**.

In **induction**, you investigate several pairs of inputs and outputs to attempt to find a pattern. That pattern can then be generalized into an algorithm.

In **top-down design**, you attempt to simplify the problem by breaking it down into multiple easier problems. You can then write code using **helper functions** that solves the main problem, then write each helper function individually to complete the work.

Example: Generate with Induction

Example: Write the function nearestBusStop(street) that takes a non-negative int street number, and returns the nearest bus stop to the given street, where buses stop every 8th street, including street 0, and ties go to the lower street, so the nearest bus stop to 12th street is 8th street, and the nearest bus stop to 13th street is 16th street.

Look at where the output changes based on the input, graph the results, and write the program.

4 – 0

5 – 8

8 – 8

12 – 8

13 – 16

Example: Generate with top-down design

Example: Write the function mostFrequentDigit(n), that takes a non-negative integer n and returns the digit from 0 to 9 that occurs most frequently in it, with ties going to the smaller digit.

This initially seems difficult- how can we keep track of the most frequent number as we go through the digits? It helps to think about it from a different angle- can we look at **each possible digit's count**?

If we assume we can write a helper function digitCount(n, d) (where n is the initial number and d is the digit we're counting), the problem becomes simpler; we just need to go from 0 to 9, calling the helper function and updating the most frequent digit as needed.

Writing digitCount is then easy- it's just a variant of numberLength!

Carrying out the Plan

Once we've come up with a working algorithmic plan, we need to **carry out the plan** by translating it into code. As you become fluent in coding and Python, you'll be able to go straight from idea to code, but when you're starting out, this might be difficult!

If you're having trouble, it can help to start by **writing the algorithm as a series of steps in plain language**, either on paper or in comments. Your written algorithm should be clear and complete enough that another person could carry it out, and should distinctly label any information that needs to be remembered (that information will become variables later on).

Once your written algorithm is complete, you can translate **one step at a time** into code. If you don't know how to do a certain step in Python, you can check the course notes or ask a TA to find the appropriate programming tool, then experiment with that tool in the interpreter until you understand it.

Reviewing your work

Finally, once your program is written, you need to **review your work** by running and testing your code. This isn't like spot-checking your work on a written assignment, because the computer can already tell you when something is wrong!

Remember the input-output pairs we considered in Step 1? We can now turn those into **test cases**, then run the program on the test cases to make sure it's working appropriately. When a test case fails, we can use **debugging** to determine what's going wrong. Once all the test cases are passing, you should read your program one more time to make sure it makes sense, then submit it for final review.

We'll go over testing and debugging in more depth next. First, let's practice problem solving...

Exercise: nthPrime

Exercise: write the function nthPrime(n), which takes a non-negative int, n, and returns the nth prime number, starting from 0 (so nthPrime(0) returns 2). A prime number is a number that only has two factors: itself and 1.

First, let's **understand the problem**.

nthPrime(0) = 2

nthPrime(1) = 3

nthPrime(2) = 5

nthPrime(3) = 7

nthPrime(4) = 11

nthPrime(-1) = not allowed!

Exercise: nthPrime

Next, let's **devise a plan**.

nthPrime can follow a common pattern we'll see on this class's problems- **nthItem**. This function will track two variables; **guess**, which keeps track of the number we're currently guessing, and **found**, which keeps track of how many numbers we've found that match the description. We'll keep looping, updating the guess number each time, until found reaches the inputted number.

nthPrime will use a helper function, **isPrime**, which will tell us whether a given number is prime. Here we can use the definition of a prime number- it must only have 1 and itself as factors. So we can **check all the intermediate numbers** to see if any of them are factors too; if none are, the number is prime.

Exercise: nthPrime

Next, we **carry out the plan**

Let's code isPrime and nthPrime together!

Exercise: nthPrime

Finally, we **review our work**.

Here are test cases based on our input/output pairs from before:

```
assert(nthPrime(0) == 2)
assert(nthPrime(1) == 3)
assert(nthPrime(2) == 5)
assert(nthPrime(3) == 7)
assert(nthPrime(4) == 11)
```

But we should also test isPrime to make sure our helper function is working first!

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:

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**, **semantic errors**, and **logical errors**.

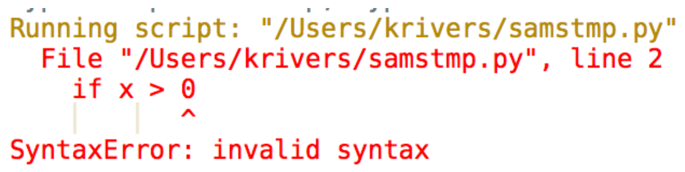
Debugging Syntax Errors

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:

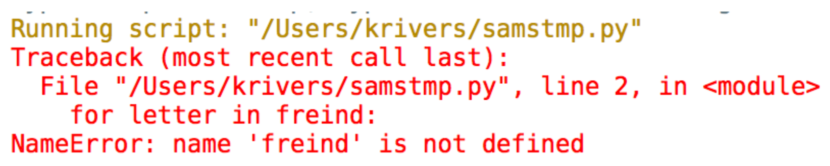
1. Read the error message and identify the type of error.
2. Look for the line number, go to that line of code, and identify which part might be associated with the error.
3. Use print statements to identify what the code's **state** is at that point in the program, and work out what the state should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

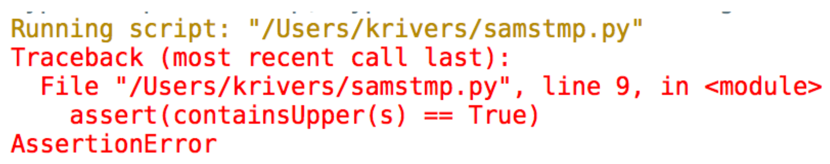
Debugging Logical Errors

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 the problem-solving process to develop solutions to new problems

Use testing and debugging to verify that our programs work