CS111 Fall 2025 Solutions for Midterm 1 Review Problems

These are solutions to the Fall 2025 Midterm 1 Review Problems. You should solve a problem **before** you look at its solution!

Table of Contents

Python Basics

Python Basics 1: Python Calisthenics

Part 1a: Examine each snippet of code below to find the value of the variable **a** and state its type. Write the value in the second column and the type in the third column. If the code evaluates to an error, write the kind of error and use the third column to briefly explain the error. *The first two have been done as examples for you.*

Please write only within the table below:

Code:	**What is the value of a (or Error)?**	**What type is a? (or Error explanation)**
`a = 3 + 4`	7	int
`a = float('3.4.5')`	Value Error	Although `float` can work on some strings, it doesn't work on `'3.4.5'`
`a = (17 // 3) * 3 + 17 % 3`	17	`int`
`b = 'cat'` `a = b[2] + b[3]`	Index Error	Index 3 does not exist in `'cat'`
`b = 'dog'` `a = b[-1] + b[1]`	`'go'`	`str`
`b = len('3.5' * 3)` `a = b * 2`	18	int

`def joy():`
`  print(3)`
`a = joy() + 17`
	Type Error	`joy()` returns `None`, which cannot be used as an operand for `+`.

Part 1b: In the box below to the right, show what is printed by the following code. Note that the `sing` function is nonsense and should **not** be assumed to produce a meaningful result.

```
def sing(coda):
    song = 3.1
    print(round(song) % 2)
    lyric = str(song)
    for i in lyric:
        print(i)
    lyric = lyric + coda
    print(lyric)

print(sing("hello"))
```

Show printed output here

```
1
3
.
1
3.1hello
None
```

Python Basics 2: Python basics

Part 2a For each program, show the **printed output** from the final program statement. If executing the program causes an error, write (1) the kind of error and (2) why it occurs (briefly!).

```
a = 4
b = str(a) * a
b + '!'
print(b)
```

Printed output:

4444

```
x = 4
y = 2 * x
z = y + 1
x = 5
print(x, y, z)
```

Printed output:

5 8 9

```
animal = "cat"
print(animal[1], animal[3])
```

Printed output:

Index Error (b/c 3 is not a valid index)

```
i = int(5.9)
f = float("3")
print(i * f)
```

Printed output:

15.0 # a float, not an int

```
s= "watermelon"
print(s[:3] + s[-1:-5:-2])
```

Printed output:

watnl

```
c = 8
d = 3
r = round(c / d)
print(r, (c // d))
```

Printed output:

3 2

Part 2b: In the box below to the right, show the **printed output** from the following code.

```
num = 4

def show(num):
    saved = num
    double = num * 2
    num = 12
    half = num / 2
    both = double + half
    print('n', num)
    print('dtb', double, half, both)
    print('s', saved)

show(10)
print('z', num)
```

Show printed output here

n 12
dtb 20 6.0 26.0
s 10
z 4

Python Basics 3: Python built-in functions

Part 3a For each program, show the output **printed** by the final program statement. If executing the program causes an error, write (1) the kind of error and (2) why it occurs (briefly!).

```
a = "23"
b = int(a) - len(a)
b*2
print(b)
```

Printed output:

21

```
s1 = "a"
s2 = "b"
s3 = s1 + s2*3
s1 = "c"
print(s3 + s1)
```

Printed output:

abbbc

```
city = "Newton"
print(city[3], city[6])
```

Printed output:

Index Error (b/c 6 is not a valid index)

```
i = int(3.7)
f = float("5")
print(i + f)
```

Printed output:

8.0 # a float, not an int

```
s = "4.9"
print(int(s) + float(s))
```

Printed output:

Value error (b/c int can't work on "4.9")

```
c = 14
d = 5
print(c//d, c%d,
      int(c/d), round(c/d))
```

Printed output:

2 4 2 3

Part 3b: In the box below to the right, show what is printed by the following code. Note that the `mystery` function is nonsense and should **not** be assumed to produce a meaningful result.

```
string = 'ABC'

def mystery(s):
    n = len(s)/2
    print(n)
    chars = str(n)
    i = int(chars[2])
    string = s.lower() + (s[1]*i)
    return string

print(mystery('DEF'))
print(string)
```

Show printed output here

1.5
defEEEEE
ABC

Python Basics 4: Python built-in functions

Part 4a For each program, show the output printed by the final program statement. If executing the program causes an error, write (1) the kind of error and (2) why it occurs (briefly!).

```
a = 11
b = 2
print(a//b, a/b)
```

Printed output:

5 5.5

```
f = float("five")
print(f+5)
```

Printed output:

ValueError: can't convert "five" to float

```
v1 = ""
v2 = " "
v3 = "cat"
print(len(v3+v2+v1))
```

Printed output:

4

```
a = "b"
c = "d"
print(a+b+c)
```

Printed output:

NameError: b is undefined

```
line1 = "a,b"
line2 = line1+"c"
print(line2)
```

Printed output:

a,bc

Part 4b: Professor Anderson is cutting strips of fabric to make a border around a quilt. In the box below, write a program (not a function) that uses the input function (twice) to ask her for the height and width of the quilt, and then prints out the total length of the fabric that she needs. Below is a sample execution of the program:

```
Enter height in feet: 8
Enter width in feet: 6
You will need 28 feet of fabric.
```

The program calculates that 28 feet will be needed to cover all sides of the quilt (8+8+6+6).

You may assume that Professor Anderson enters valid dimensions when asked.

Write your Part 1b program in this box:

```
a = float(input("Enter height in feet: "))
b = float(input("Enter width in feet: "))
perimeter = 2 * a + 2 * b
print("You will need", perimeter, "feet of fabric.")
```

Python Basics 5: Python built-in functions

Part 5a: For each program, show the output printed by the final program statement. If executing the program causes an error, write (1) the kind of error and (2) why it occurs (briefly!).

```
a = 23
b = 4
print(a//b, a%b,
round(a/b))
```

Printed output:
5 3 6

```
f = float("5.6")
i = str(37)
print(f + i)
```

Printed output:
TypeError: can't add float & string

```
x = "12"
y = int(x) - len(x)
y*2
print(y)
```

Printed output:
10

```
color1 = "blue"
color2 = "red"
color3 = 2 * color1 +
color2
color1 = "cyan"
print(color3 + color1)
```

Printed output:
bluebluereddcyan

```
val1 = "one"
val2 = "two"
sum = int(val1) + int(val2)
print(sum)
```

Printed output:
ValueError: can't convert "one" to int

Part 5b: The **harmonic mean** of two numbers is calculated through the formula:

$$h = \frac{2ab}{a+b}$$ in other words, *h* is (2 times a times b), divided by (a plus b)

In the box below, write a program (**not** a function) that uses the `input` function (twice) to prompt the user to enter numbers for **a** and **b**, calculates the harmonic mean and stores it in a variable **h**, and then prints out the message:

`The harmonic mean for` *a* `and` *b* `is` *h*

where *a* and *b* are the values of the two numbers entered by the user and *h* is the calculated harmonic mean. Below is a sample execution of this program:

```
Enter number a: 10
Enter number b: 2.5
The harmonic mean of 10.0 and 2.5 is 4.0
```

You may assume that all inputs are numbers.

Write your harmonic mean program in this box:

```
a = float(input("Enter number a: "))
b = float(input("Enter number b: "))
h = 2 * a * b / (a + b)
print("The harmonic mean of", a, "and", b, "is", h)
```

Simple Functions

Simple Functions 1: Defining a `repeatIt` function

In the box below, define a function named `repeatIt` that takes a single parameter that is a string, calls the `input` function to get an integer from the user, and **return**s the string repeated as many times as the user specified. You may assume that the user will always enter an integer. You may also assume that when the function `repeatIt` is called that the argument passed is a string.

```
def repeatIt(text):
    times = input("Enter an integer: ")
    return int(times)*text
```

Simple Functions 2: Functions with return and print

```
def red(word):
    newWord =  'red' + word
    print(word) # this is the only function with print in it
    return newWord

def blue(word):
    newWord = 'blue' + word
    return newWord

def green(word):
    newWord = 'green' + word
    return newWord
```

Part 5a. Given the function definitions above, what is printed by the following code?

```
newWord = 'grouch'
print(green(red(blue('oscar'))))
print(newWord)
```

Write what is printed in the box below

```
blueoscar            # the argument, not the newWord
greenredblueoscar    # the final result
grouch               # nothing changes the global value of newWord
```

Part 5b. Suppose we remove the **print** invocation in the `red` function. Then all three functions above are very similar. Write a function called `addColor` that captures the pattern in the `red`, `blue` and `green` functions above. Your `addColor` function should have two string arguments and return a string (see examples below):

```
addColor('orange','cake') ⇒ 'orangecake'
addColor('purple','eyes') ⇒ 'purpleeyes'
```

Define your function in the box below

```
def addColor(color, word):
    return color + word
```

Simple Functions 3: Custom Functions

Part 3a Vocabulary

```
# line 1   def swift(anti, hero):
# line 2       greet = "hi!" * anti
# line 3       refrain = "it's me"
# line 4       return refrain + greet + hero
# line 5
# line 6   swift(3,"I")
# line 7   swift(5,"you")
```

3a(i): Identify all **function parameters** above:

Parameter	Line #
anti	line 1
hero	line 2

3a(ii): Identify all **function arguments** above:

Argument	Line #
3	line 6
"I"	line 6
5	line 7
"you"	line 7

3a(iii) What is the difference between a parameter and an argument? (1-3 sentences)

A **parameter** is a variable name in the header of a function definition that is used in the function body to stand for the argument value that is supplied when the function is called.

An **argument** is a Python value that will effectively replace the corresponding parameter name in the body of the function when it is called.

In the function frame model, each parameter names a variable box whose contents are initialized to the argument value in the call.

Part 3b Define the nameRow function

In the box below, define a function called `nameRow` that has no parameters and does not return anything. It should call the `input` function twice: once to get the user's name, and once to get a number indicating how many times the name should be repeated. It should print the name on one line the requested number of times.

Assume that the user will enter a positive integer when prompted. **For full credit, your solution must not use any loops.** Example printed output when the user's name is Carolyn and the user requests a row of length 2: `CarolynCarolyn`

```
def nameRow():
    name = input("Enter your name: ")
    repeats = int(input("Enter the number of times to repeat name: "))
    print(name*repeats)
```

Simple Functions 4: Custom Functions

```
def someLaughter():
      print('LAUGHTER')
      print('LAUGHTER')
      print('LAUGHTER')
      print('LAUGHTER')
      print('LAUGHTER')
```

You are given the function `someLaughter` above that prints `'LAUGHTER'` five (5) times. You must define two **zero-parameter** functions, one of which is named `megaLaughter` and a helper function whose name you choose. When called on zero arguments, `megaLaughter` should print `'LAUGHTER'` one hundred (100) times.
For full credit, your solution must meet all these criteria:

- Define another helper function that calls `someLaughter`
- `megaLaughter` does not call `someLaughter` directly
- Neither `megaLaughter` nor the helper function may call `print` directly
- `megaLaughter` contains no more than 5 lines of code in the function body (not counting the `def` line)
- There are no loops anywhere in your solution, including in your helper function

You will receive partial credit if your code prints `'LAUGHTER'` one hundred (100) times even if it doesn't meet all of the criteria above.

Write your solution in this box

```
# Solution #1
def moreLaughter():
     someLaughter()
     someLaughter()
     someLaughter()
     someLaughter()

def megaLaughter():
     moreLaughter()
     moreLaughter()
     moreLaughter()
     moreLaughter()
     moreLaughter()
```

```
# Solution #2
def moreLaughter():
     someLaughter()
     someLaughter()
     someLaughter()
     someLaughter()
     someLaughter()

def megaLaughter():
     moreLaughter()
     moreLaughter()
     moreLaughter()
     moreLaughter()
```

Simple Functions 5: Defining and calling functions

Part 5a: Consider the following three functions:

```
def a():
    return 'a'
```

```
def b(s):
    return s + 'b'
```

```
def c(s):
    return s + 'c' + s
```

In the table below, fill in the results for each expression (write your answers as quoted strings).

Expression consisting only of calls to the three functions above	String that is the value of the expression
`b(c(a()))`	`'acab'`
`c(b(b(a())))`	`'abbcabb'`

Part 5b: You are given the following function definition for `printCombinations`:

```
def printCombinations(x, y, z):
    plus = x + y
    times = y * z
    print(plus, y, times)
```

In the box below, fill in the missing arguments to the `printCombinations` function calls so that `printNums()` **prints** this output:

```
5 4 12
7 2 8
```

```
def printNums():
    printCombinations(1, 4, 3 )
    printCombinations(5, 2, 4 )
```

Part 5c : Define a function named pickNums that takes three **integers** and **prints** them right-justified with brackets to their left as shown in these examples:

`pickNums(3, 4, 15)` **prints**
`[ ]  3`
`[ ]  4`
`[ ] 15`
	`pickNums(9231, 2, 950)` **prints**
	`[ ] 9231`
	`[ ]    2`
	`[ ]  950`

Your definition **must not** include any conditionals or loops and **must** include **three** invocations of the following `optRow` function (in addition to other function calls):

```
def optRow(num, indent):
     return '[ ] ' + (' ' * indent) + str(num)
```

Define your **pickNums** *function in this box:*

```
def pickNums(a, b, c):
    sa = len(str(a))
    sb = len(str(b))
    sc = len(str(c))
    maxLen = max(sa, sb, sc)
    print(optRow(a, maxLen - sa))
    print(optRow(b, maxLen - sb))
    print(optRow(c, maxLen - sc))
```

Simple Functions 6: Defining and calling functions

Part 6a: Consider the following three functions:

```
def one():
    return 1
```

```
def dbl(n):
    return 2*n
```

```
def incDbl(n):
    return 1 + (2*n)
```

It turns out that any positive integer can be expressed using nested calls to just these three functions. In the table below, fill in the missing parts.

Expression consisting only of calls to the three functions one, dbl, and incDbl	Positive integer that is the value of the expression
`incDbl(dbl(dbl(one())))`	9
`dbl(incDbl(incDbl(one())))`	14

Part 6b: You are given the following function definition for `printPatternLine`:

```
def printPatternLine(char1, char1Repeat, char2, chunkWidth, chunkRepeat):
    chunk = (char1*char1Repeat) + (char2*(chunkWidth - char1Repeat))
    print(chunk*chunkRepeat)
```

In the box below, fill in the missing arguments to the `printPatternLine` function calls so that `printPattern()` **prints** this output:

```
Y.....Y.....Y.....
ZZZZ-----ZZZZ-----
```

```
def printPattern():

    printPatternLine('Y', 1, '.', 6, 3                    )

    printPatternLine('Z', 4, '-', 9, 2                    )
```

Part 6c: In the box below, define a function named box3 that takes three strings and **prints** them left-justified inside a rectangular box made of +, -, and | characters, as shown in these examples:

box3('apple', 'banana', 'pear') **prints**	box3('two', 'roads', 'diverged') **prints**
+------+ |apple | |banana| |pear | +------+	+--------+ |two | |roads | |diverged| +--------+

Your definition **must not** include any conditionals and **must** include **three** invocations of the following `printBoxLine` function (in addition to invocations of `print`):

```
def printBoxLine(word, numSpaces):
     print('|' + word + (' '*numSpaces) + '|')
```

Define your **box3** *function in this box:*

```
def box3(s1, s2, s3):
    maxLen = max(len(s1), len(s2), len(s3))
    line = '+' + '-' * maxLen + '+'
    print(line)
    printBoxLine(s1, maxLen-len(s1))
    printBoxLine(s2, maxLen-len(s2))
    printBoxLine(s3, maxLen-len(s3))
    print(line)
```

Simple Functions 7: Defining and calling functions

Part 7a: Define a function `rectangle` that takes four parameters — two strings that are characters (a border character and a filler character) and two integers (the width and the height of the rectangle) — and **prints** rectangles like the ones shown below.

- Assume that each of the width and the height is 2 or greater.
- Recall that * can be used to repeat a string.
- For **full credit**, your function **must** use a **`while`** loop (but **substantial partial credit** will be awarded if it uses a correct **`for`** loop).

`rectangle('#', '.', 5, 4)`	`rectangle('@', '-', 6, 3)`	`rectangle('&', '+', 4, 6)`

`#####`
`#...#`
`#...#`
`#####`
	`@@@@@@`
	`@----@`
	`@@@@@@`
		`&&&&`
		`&++&`
		`&++&`
		`&++&`
		`&++&`
		`&&&&`

Define your `rectangle` function in this box.

```
def rectangle(border, filler, width, height):
    # Top border
    print(border * width)

    # Middle
    middleReps = height - 2;
    while middleReps > 0:
        print(border + filler * (width - 2) + border)
        middleReps -= 1
    # for loop version:
    # for _ in range(height - 2):
    #   print(border + filler * (width - 2) + border)

    # Bottom border
    print(border * width)
```

Part 7b: (You do not have to define `rectangle` correctly in part 3a in order to answer this part.) In the box below, write an invocation of the `rectangle` function that will display the pattern shown in the box to the left (assuming `rectangle` is correctly defined).

```
????????
?      ?
?      ?
?      ?
????????
```

Write your invocation of rectangle in this box:

```
rectangle('?', ' ', 8, 5)
```

Simple Functions 8: Defining and calling functions

Define a function `buildSandwich` that takes three parameters: bread (a string), filling (a string), and layers (an integer). Your function should **print** lines of text to form a sandwich: the bread string appears on the top and bottom with the filling word in the middle. The total number of lines should be the number of layers requested by the user. Your function should use the printFilling helper function defined below to do this, and should NOT include any loops.

```
def printFilling(f,n):
   for i in range(n):
      print(f)
```

`> buildSandwich('pita','falafel',3)`	`> buildSandwich('toast','butter',5)`	`> buildSandwich('rye','tuna',4)`
pita falafel pita	toast butter butter butter toast	rye tuna tuna rye

Define your `buildSandwich` *function in this box.*

```
def buildSandwich(bread, filling, layers):
    print(bread)
    printFilling(filling, layers-2)
    print(bread)
```

Booleans and Predicates

Booleans and Predicates 1: `exactlyTwoEqual` predicate

In the box below, define a function named `exactlyTwoEqual` that takes three numbers and returns `True` if exactly two or them are equal and `False` otherwise. For example:

```
exactlyTwoEqual(6, 8, 6) ⇒ True    exactlyTwoEqual(6, 8, 5) ⇒ False
exactlyTwoEqual(7, 7, 4) ⇒ True    exactlyTwoEqual(5, 5, 5) ⇒ False
exactlyTwoEqual(8, 9, 9) ⇒ True
```

```
# Solution 1
def exactlyTwoEqual(n1, n2, n3):
  return ((n1 == n2 and n2 != n3) # can also be written (n1 == n2 != n3)
          or (n1 == n3 and n1 != n2) # can also be written (n2 != n1 == n3)
          or (n2 == n3 and n1 != n2) # can also be written (n1 != n2 == n3)
          )

# Solution 2
def exactlyTwoEqual(n1, n2, n3):
  return (((n1 == n2) or (n2 == n3) or (n1 == n3))
          and not ((n1 == n2) and (n2 == n3))
                   # can also be written (n1 == n2 == n3)
                   )
```

Booleans and Predicates 2: Age Predicates

In this problem you will define and use **predicates**, which are functions that return booleans. **You are NOT allowed to use `if/else` statements in any of your definitions. Instead, you should combine booleans with `and/or/not`.**

Part 7a: Define a predicate `isTeenager` that has one parameter for age (an integer) and returns true when the age is in the teen years (thirteen to nineteen). For example, `isTeenager(13)` and `isTeenager(19)` should both return `True`, but `isTeenager(12)` and `isTeenager(20)` should both return `False`.

```
def isTeenager(age):
    return age >= 13 and age <= 19
    # or: 13 <= age <= 19
```

Part 7b:: Assume that you have been given correct definitions for the following two predicates:

- `isMinor(age)`: returns `True` if `age` <= 15, and `False` otherwise.
- `canRetire(age)`: returns `True` if `age` >= 67, and `False` otherwise.

Define a predicate `isWorkingAge` that has one parameter for age and returns `True` if a person with that age is of working age (between the ages of 16 and 66, inclusive) and `False` otherwise. Your definition **must not contain any numbers**. Instead, it **must call both the `isMinor` and `canRetire` functions to determine the answer**. For example, `isWorkingAge(16)` and `isWorkingAge(66)` should both return `True`, but `isWorkingAge(15)` and `isWorkingAge(67)` should both return `False`.

```
def isWorkingAge(age):
    return not(isMinor(age) or canRetire(age))
    # or: return not isMinor(age) and not canRetire(age)
```

Part 7c:: Define a predicate `isNonWorkingAge` that has one parameter for the age and returns `True` if a person with that age is **not** of working age (as defined above) and `False` otherwise. Your definition **must not contain any numbers and must *not* call `isWorkingAge`**. Instead **must call both the `isMinor` and `canRetire` functions to determine the answer**. For example, `isNonWorkingAge(15)` and `isNonWorkingAge(67)` should both return `True`, but `isNonWorkingAge(16)` and `isNonWorkingAge(66)` should both return `False`.

```
def isNonWorkingAge(age):
    return isMinor(age) or canRetire(age)
```

Part 7d: Define a predicate `isWorkingTeenager` that has one parameter for the age and returns `True` if a person with that age is a teenager who is of working age and `False` otherwise. Your definition **must not contain any numbers and must *not* call `isMinor` or `canRetire`.** Instead it **must call both `isTeenager` and `isWorkingAge`** (which you can assume are correct). For example, `isWorkingTeenager(16)` and `isWorkingTeenager(19)` should both return `True`, but `isWorkingTeenager(15)` and `isWorkingTeenager(20)` should both return `False`.

```
def isWorkingTeenager(age):
    return isTeenager(age) and isWorkingAge(age)
```

Booleans and Predicates 3: Understanding and Defining Predicates

Part 3a: The following mysteryPred predicate takes three boolean arguments and returns a boolean result.

```
def mysteryPred(bool1, bool2, bool3):
    return ((bool1 or bool2 or bool3)
            and (not (bool1 and bool2 and bool3))
```

Fill in the following table to show the results of calls to the `mysteryPred` function:

Function call	Result		Function call	Result
`mysteryPred(False, False, False)`	`False`		`mysteryPred(True, False, True)`	True
`mysteryPred(True, False, False)`	`True`		`mysteryPred(True, True, True)`	`False`

Part 3b: Define a predicate named `isShortIn` that takes two string arguments `s1` and `s2` and returns `True` only if all **three** of the following conditions are satisfied:

1. `s1` is a substring in `s2`
2. `s1` has at most three characters.
3. `s2` does not begin with the substring `s1`. (You can use **string slicing** to test this!)

For example:

Function call	Result		Function call	Result
`isShortIn('war', 'toward')`	`True`		`isShortIn('it', 'kitty')`	`True`
`isShortIn('ward', 'toward')`	`False`		`isShortIn('kit', 'kitty')`	`False`
`isShortIn('to', 'toward')`	`False`		`isShortIn('dog', 'kitty')`	`False`

For **full** credit, your definition should not use any conditionals (`if` statements).

Define your isShortIn predicate in this box:

```
def isShortIn(s1, s2):
    return ((s1 in s2)
            and len(s1) <= 3
            and not s1 == s2[:len(s1)])
```

Booleans and Predicates 4: Predicates

Part 4a: Define a predicate named `outsideRange` that takes three numbers (`num`, `lo`, and `hi`), where you may assume that `lo` is less than or equal to `hi`. It **returns** `True` when `num` is outside the range between `lo` and `hi` (inclusive) and `False` otherwise. For example:

```
outsideRange(1, 3, 5) ⇒ True   outsideRange(2, 3, 5) ⇒ True   outsideRange(3, 3, 5) ⇒ False
outsideRange(4, 3, 5) ⇒ False  outsideRange(5, 3, 5) ⇒ False  outsideRange(6, 3, 5) ⇒ True
```

Below, complete the two different function definitions for `outsideRange` so that they both behave correctly.

```
def outsideRange(num, lo, hi):
    return      num < lo      or      hi < num
```

```
def outsideRange(num, lo, hi):
    return not (      lo <= num     and      num <= hi      )
```

Part 4b(i): The three most frequent letters in English texts are `e`, `t`, and `a`. In the box below, define a predicate named `isFrequentLetter` that takes a single string argument. It **returns** `True` if the string is a lower-case or upper-case version of one of these three letters, and `False` otherwise. For example:

```
isFrequentLetter('e') ⇒ True   isFrequentLetter('T') ⇒ True   isFrequentLetter('a') ⇒ True
isFrequentLetter('x') ⇒ False  isFrequentLetter('B') ⇒ False  isFrequentLetter('eta') ⇒ False
```

Recall that if `s` is a string, then `s.lower()` returns the lower-case version of the string.

```
# In this definition, you must *not* use any conditional (if/else) statements
def isFrequentLetter(char):
    low = char.lower()
    return low == 'e' or low == 't' or low == 'a'
    # or: return len(char) == 1 and char.lower() in 'eta'
```

Part 4b(ii): In the box below, define a predicate named `containsAllFrequentLetters` that takes a single string argument. It **returns** `True` if the string contains **all** of the letters `e`, `t`, and `a` in any case (lower or upper) and `False` otherwise. For example:

```
containsAllFrequentLetters('cattle') ⇒ True   containsAllFrequentLetters('eagle') ⇒ False
containsAllFrequentLetters('TEAM') ⇒ True     containsAllFrequentLetters('CS111') ⇒ False
```

```
# In this definition, you must *not* use any conditional (if/else) statements
def containsAllFrequentLetters(word):
    low = word.lower()
    return 'e' in low and 't' in low and 'a' in low
```

Booleans and Predicates 5: Predicates

In this problem you will define and use **predicates**, which are functions that return booleans. **You are NOT allowed to use `if/else` statements in any of your definitions. Instead, you should combine booleans with `and/or/not`.**

Part 5a [2 pts]: Define a predicate `isCurrentStudent` that has one parameter for class year (an integer) and returns `True` when the year is the graduation date of a current college student (2024-2027). For example, `isCurrentStudent(2023)` and `isCurrentStudent(2028)` should both return `False`, but `isCurrentStudent(2024)` and `isCurrentStudent(2027)` should both return `True`.

```
def isCurrentStudent(classYear):
    return 2024 <= classYear <= 2027
    # or return 2024 <= classYear and classYear <= 2027
```

Part 5b [2 pts]: Define a predicate `isRedClass` that has one parameter for class year and returns `True` if that class year's color is red and `False` otherwise. The class color of 2024 is red; class colors rotate on a 4 year cycle.
For example, `isRedClass(2020), isRedClass(2024)` and `isRedClass(2028)` should all return `True`, but `isRedClass(2025)` should return `False`.

```
def isRedClass(classYear):
    return classYear%4 == 0
```

Part 5c [4 pts]: Assume that you have been given correct definitions for the following two predicates:

- `isPurpleClass(year)`: returns `True` if the class year was a purple class (2014, 2018, 2022, 2026 …) and `False` otherwise.
- `isAlum(year)`: returns `True` if the graduation year is before 2024 and `False` otherwise.

Define a predicate `isPurpleAlum` that has one parameter for year and returns `True` if a person with that graduation year is an alum from a purple class and `False` otherwise. Your definition **must not contain any numbers**. Instead, it **must call both the `isPurpleClass` and `isAlum` functions to determine the answer**. For example, `isPurpleAlum(2022)` and `isPurpleAlum(2018)` should both return `True`. `isPurpleAlum(2026)` and `isPurpleAlum(2023)` should both return `False`.

```
def isPurpleAlum(classYear):
    return isPurpleClass(classYear) and isAlum(classYear)
```

Part 5d [2 pts]: Define a predicate `isNotPurpleAlum` that has one parameter for the class year and returns `True` if a person with that class year is **not** of a purple class alum and `False` otherwise. Your definition **must not contain any numbers and must *not* call `isPurpleAlum`**. It **must call both the `isAlum` and `isPurpleClass` functions to determine the answer**. For example, `isNotPurpleAlum(2026)` and `isNotPurpleAlum(2023)` should both return `True`, but `isNotPurpleAlum(2022)` and `isNotPurpleAlum(2018)` should both return `False`.

```
def isNotPurpleAlum(classYear):
    return (not isPurpleClass(classYear)) or (not isAlum(classYear))
```

Conditionals

Conditionals 1: Understanding conditionals

In the table below, show what is printed for various calls of this **`analyze`** function:

```
def analyze(word):
    if len(word) <= 4:
        print('S')
    else:
        print('L')
    if isVowel(word[0]):
        print('V0')
        if not isVowel(word[1]):
            print('C1')
    elif isVowel(word[1]):
        print('V1')
    else:
        print('C01')
    if isVowel(word[-1]): # last letter of word
        print('VU')
        if not isVowel(word[-2]): # next to last letter of word
            print('CP')
```

```
def isVowel(char):
    return len(char) == 1 and char.lower() in 'aeiou'
```

Function call	Printed Output		Function call	Printed Output
`analyze('cat')`	S V1		`analyze('spree')`	L C01 VU
`analyze('oats')`	S V0		`analyze('apple')`	L V0 C1 VU CP

Conditionals 2: Conditionals with `whichName`

Define a function named `whichName`, which takes two parameters that represent potential cat names and returns which one is best. The function `whichName` must **return** the best cat name indicated as the string `'#1'` or `'#2'` given the following rules:

- Names with titles ("Mr. Biggles") are the best. Any string that includes a period contains a title.
- If both names have a title or neither name has a title, the longer name is best.
- If the length of the names are the same, choose the name that is alphabetically last.

Examples:

```
In[]: whichName("Ms. Piggy","Whiskers")
Out[]: '#1'

In[]: whichName("Fancy Feast","Giganotosaurus")
Out[]: '#2'

In[]: whichName("Tuna","Foxy")
Out[]: '#1'

In[]: whichName("Ms. Piggy","Ms. Puffy")
Out[]: '#2'
```

You must not use loops for this problem. Hint: < and > can be used to compare strings.

```
# Define your whichName function in this box

def whichName(name1, name2):
    # First, choose a title over a non-title:
    if '.' in name1 and '.' not in name2:
        return '#1'
    elif '.' not in name1 and '.' in name2:
        return '#2'

    # Get here only if both names are titles or neither is a title.
    # In this case find the longer one.
    elif len(name1) > len(name2):
        return '#1'
    elif len(name1) < len(name2):
        return '#2'

    # Get here only if there's no answer yet and both names
    # have the same length. In this case return the one that's
    # alphabetically last.
    elif name1 > name2:
        return '#1'
    else:
        return '#2'
```

Conditionals 3: Printing Time (Function with Conditionals & Booleans)

In the box at the bottom of this problem, define a function **`printTime`** that takes three arguments:

1. **`day`**: a day of the week, which is one of the strings **`'Sun'`**, **`'Mon'`**, **`'Tue'`**, **`'Wed'`**, **`'Thu'`**, **`'Fri'`**, **`'Sat'`**
2. **`hour`**: an integer between 1 and 12, inclusive
3. **`ampm`**: one of the strings **`'AM'`** or **`'PM'`**

printTime prints *exactly one word* as specified below. It does not return anything.

- For a weekend day (Sat or Sun), it prints **`weekend`**.
- For a weekday (Mon through Fri):
  - It prints **`evening`** from 5PM up to and including 11PM
  - It prints **`sleep`** from midnight (12AM) up to and including 8AM.
    Note that midnight is considered the beginning of a new day, not the end of a previous day.
  - It prints **`class`** for all other times — i.e., from 9AM up to and including 4PM.
    This range includes noon (12PM).

Here are some examples:

Function call	Printed Output		Function call	Printed Output
`printTime('Sat',12,'AM')`	`weekend`		`printTime('Mon',12,'AM')`	`sleep`
`printTime('Sat',10,'AM')`	`weekend`		`printTime('Wed',3,'AM')`	`sleep`
`printTime('Sun',11,'PM')`	`weekend`		`printTime('Fri',8,'AM')`	`sleep`
`printTime('Mon',5,'PM')`	`evening`		`printTime('Tue',9,'AM')`	`class`
`printTime('Thu',8,'PM')`	`evening`		`printTime('Wed',12,'PM')`	`class`
`printTime('Fri',11,'PM')`	`evening`		`printTime('Thu',4,'PM')`	`class`

In your definition you do **not** need to handle cases where an input is an unexpected value (e.g., an invalid day or ampm string or an hour that is not an integer in the range 1 to 12 inclusive).

(Please keep all your code within the box)

```
def printTime(day, hour, ampm):
    if day == "Sat" or day == "Sun":
        print("weekend")
    elif ampm == "PM" and 5 <= hour and hour <= 11:
        # Alternatively can write: 5 <= hour <= 11:
        print("evening")
    elif ampm == "AM" and (hour == 12 or hour <= 8): # 12AM is special case
        print("sleep")
    else:
        # Although it's not needed (since ELSE catches everything else)
        # we could use this explicit test instead for this case:
        # ((ampm = "AM" and 9 <= hour <= 11)
        #  or (ampm = "PM" and (hour == 12 or hour <= 4)))
        print("class")
```

Conditionals 4: `lmnop`

Define a function named `lmnop` that takes a single letter and returns an integer according to these rules:

- If the letter is one of the five letters in `l`, `m`, `n`, `o`, or `p` (either lower or upper case) then 5 is returned
- If the letter comes before the `l` in the alphabet (letter "el", not the digit 1!) in the alphabet, 1 is returned
- If the letter comes after `p` in the alphabet, then 3 is returned

`lmnop` should treat upper and lower case letters the same way. You may assume the input is a string consisting of a single alphabetic letter; you should **not** handle input strings whose length is not 1, nor nonalphabetic characters like digits, punctuation or spaces. Below are some sample invocations. Recall that characters can be compared alphabetically using < and >, e.g. `('a' < 'b')` is `True` because `'a'` comes before `'b'` in the alphabet.

```
 In[]: lmnop('a')
Out[]: 1
 In[]: lmnop('p')
Out[]: 5
 In[]: lmnop('L')
Out[]: 5
 In[]: lmnop('T')
Out[]: 3
 In[]: lmnop('C')
Out[]: 1
```

Define your lmnop function in this box

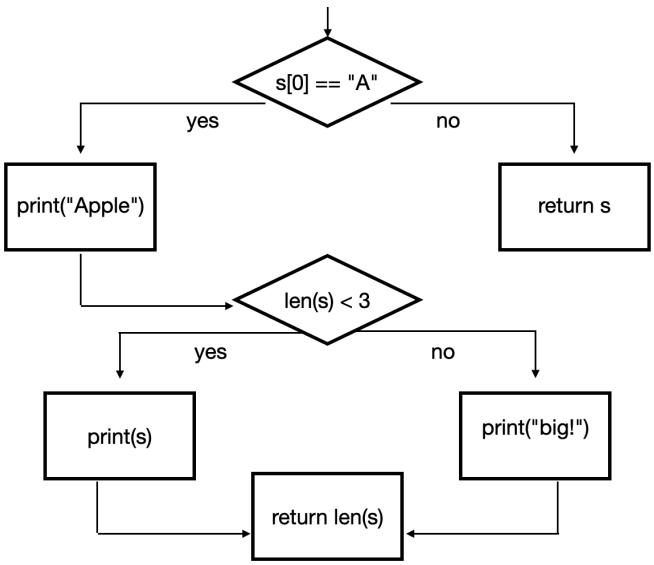
```
def lmnop(letter):
    lowerLetter = letter.lower()
    if lowerLetter in 'lmnop':
        return 5
    elif lowerLetter < 'l': # this is the letter l
        return 1 # this is the number 1
    else: # letter comes after 'p'
        return 3
```

Conditionals 5: Implementing a program based on a flow chart

Define a function called `stringExplorer` that implements the flow chart shown to the right. Your function should have one parameter, s.

Define your `stringExplorer` *function in this box:*

```
def stringExplorer(s):
    if s[0] == 'A'
        print('Apple')
        if len(s) < 3:
           print(s)
        else:
           print('big')
        return len(s)
    else:
        return s
```



Understanding `while` Loops

Understanding while Loops 1: mystery while loop

Study the **`mystery`** function below, which uses the provided **`isVowel`** function.

```
def isVowel(char):
    return len(char) == 1 and char.lower() in 'aeiou'

def mystery(word, bound):
    """Docstring withheld."""
    result = ''
    i = 0

    while len(result) < bound and i < len(word):
        if (not isVowel(word[i])) and word[i] not in result:
            result += word[i]
        i += 1

    if result == '':
        return 'No result'

    return result
```

Predict the outcome of the following invocations of the **`mystery`** function:

Function call	Value returned by function call
`mystery('pineapple', 1)`	`'p'`
`mystery('pineapple', 4)`	`'pnl'`
`mystery('guava', 2)`	`'gv'`
`mystery('oooooh', 2)`	`'h'`
`mystery('ooooo', 2)`	`'No result'`

Understanding while Loops 2: While Loops with user input

Consider this **`askForFruit`** function:

```
def askForFruit():
  name = ''
  while len(name) <= 6:
    name = input('Fruit? ')
  print('Done')
```

Select **all** the valid possible outcomes consistent with executing `askForFruit()`

✅ Choice A Fruit? Apple Fruit? Strawberry Done	✅ Choice B Fruit? LightBlue Done
❌ Choice C Fruit? Apple Done	❌ Choice D Fruit? Banana Fruit? Fofana Done
❌ Choice E Fruit? Watermelon Fruit? Apple Fruit? Grapes Done	✅ Choice F Fruit? Apple Fruit? Orange Fruit? Cantaloupe Done

Understanding while Loops 3: Using an iteration table to understand a loop

You are given this definition of a `mysteryLines` function.

```
def mysteryLines(c, h):
    i = 1
    while i < h + 1:
        s = h - i
        if i == 1 or i == h:
            m = 2 * i - 1
            line = ('-' * s) + (c * m) + ('-' * s)
        else:
            m = 2 * i - 3
            line = ('-' * s) + c + ('-' * m) + c + ('-' * s)
        # In the iteration table, show the values of state variables at this point
        print(line)
        i += 1
```

For the invocation **`mysteryLines('*', 4)`**, in each row of the iteration table below, show the values of the state variables in each execution of the **body** of the **`for`** loop **right before the call to `print`**.

- The iteration table has more rows than needed, so at least one will be blank
- Unlike some other iteration tables you have seen in class. this iteration table **should not** have any row containing the values of state variables **before** the loop is entered.

Fill in this iteration table for ***`mystery('*', 4)`***

i	h	s	m	line
1	4	3	1	'---*---'
2	4	2	1	'--*-*--'
3	4	1	3	'-*---*-'
4	4	0	7	'*******'

Understanding while Loops 4: Tracing loops and conditionals

Below is a function `doSomething` that contains a `while` loop and conditional statements. Trace the execution of invoking the `doSomething` function with different arguments by showing what is **printed** and what is **returned** for each invocation.

```
def doSomething(n):
    # n is an *integer*
    answer = ''
    # answer is a *string*
    while n > 2:
        answer = answer + str(n)
        if n%2 == 0:
            print(n, 'E')
        elif n % 9 == 0:
            print(n, 'T')
            # early return
            return answer

        if n == 10:
            print(n, 'R')
        elif n >= 7:
            print(n, 'H')
            if n <= 12:
                print(n, 'L')
            else:
                print(n, 'M')
        else:
            print(n, 'S')
        # update n in loop:
        n = n - 4
    # return result after loop
    if len(answer) >= 3:
        return '#' + answer
    else:
        return '!' + answer
```

```
>>> doSomething(10)
```

Show what is **printed**:	Show what is **returned**:
10 E 10 R 6 E 6 S	'#106'

```
>>> doSomething(13)
```

Show what is **printed**:	Show what is **returned**:
13 H 13 M 9 T	'139'

```
>>> doSomething(7)
```

Show what is **printed**:	Show what is **returned**:
7 H 7 L 3 S	'!73'

Understanding while Loops 5: Tracing loops and conditionals

Below is a function `process` that contains a `while` loop and conditional statements. Trace the execution of invoking the `process` function with different arguments by showing what is **printed** and what is **returned** for each invocation.

```
def process(n):
    # n is an *integer*
    answer = ''
    # answer is a *string*
    while n > 0:
        answer = answer + str(n)
        if n%2 == 0:
            print(n, 'E')
        if n > 9:
            print(n, 'G')
        elif n == 8:
            print(n, 'R')
            # early return
            return answer
        elif n >= 5:
            print(n, 'H')
            if n <= 7:
                print(n, 'L')
            else:
                print(n, 'M')
        else:
            print(n, 'S')
        # update n in loop:
        n = n - 5
    # return result after loop
    if len(answer) >= 3:
        return '#' + answer
    else:
        return '!' + answer
```

```
>>> process(10)
```

Show what is **printed**:	Show what is **returned**:
10 E 10 G 5 H 5 L	'#105'

```
>>> process(9)
```

Show what is **printed**:	Show what is **returned**:
9 H 9 M 4 E 4 S	'!94'

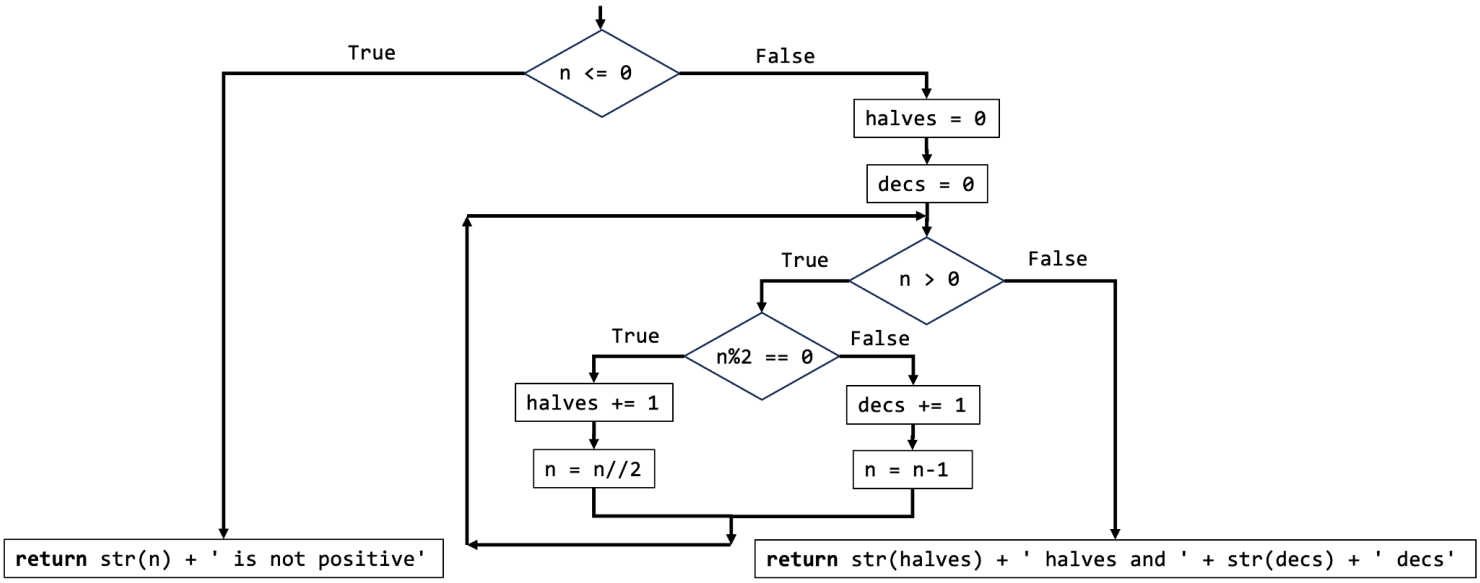
```
>>> process(8)
```

Show what is **printed**:	Show what is **returned**:
8 E 8 R	'8'

Understanding while Loops 6: Flow diagrams, iteration tables, and while loops

This problem involves a function named `halvesAndDecs`, which has a single integer parameter `n` and returns a string. The function counts the number of **halves** (n//2 operations) and **decs** (n-1 operations) performed in a loop within the body of the function. ("dec" is short for "decrement", which means to subtract 1 from a number.)

The body of the `halvesAndDecs` function is expressed by this flow diagram:



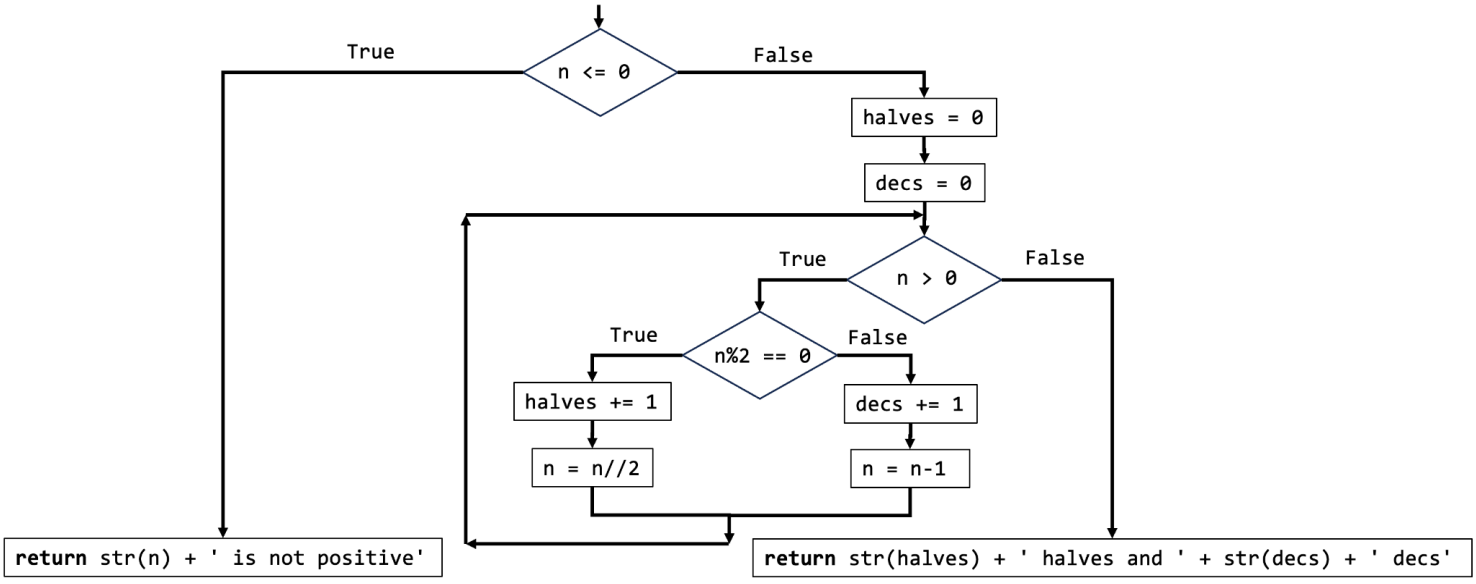
Part 6a [10 pts]: For the function invocation `halvesAndDecs(10)` , fill in the missing values in the rows of the iteration table below. Each row shows the values of the variables n, halves, and decs right before the loop condition `n > 0` is tested. **The iteration table has more rows than needed, so at least one will be blank**.

Number of times loop body has been executed	**n**	**halves**	**decs**
0	**10**	**0**	**0**
1	**5**	**1**	**0**
2	**4**	**1**	**1**
3	**2**	**2**	**1**
4	**1**	**3**	**1**
5	**0**	**3**	**2**

Part 6b [8 pts]: In the box below, complete the body of the `halvesAndDecs` function in Python so that it correctly expresses the meaning of the flow diagram (copied below). **Make sure that your indentation is clear!**

```
def halvesAndDecs(n):
    if n <= 0:
        return str(n) + ' is not positive'
    else:
        halves = 0
        decs = 0
        while n > 0:
            if n%2 == 0:
                halves += 1
                n = n//2
            else:
                decs += 1
                n = n-1
        return (str(halves) + ' halves and '
                + str(decs) + ' decs')
```

Here is a copy of the flow diagram for reference:



Understanding for Loops

Understanding for loops 1: Tracing conditionals

Given the function `calcPoints` below, show what is (1) returned and (2) printed by the following invocations. Assume that `lmnop` works correctly, as described above in Conditionals 4: `lmnop`.

```
def calcPoints(word):
    points = 0
    for char in word:
        if char == 'y':
            points = 10  # =, not +=
        elif char in 'aeiou':
            points = points * 2
        elif char in '0123456789':
            # early return
            return points + int(char)
        points += lmnop(char)
        print(char, points)
    return points
```

`In[]: calcPoints('bye')` `Out[]:` *Show what is returned:* 27	*Show what is printed:* b 1 y 13 e 27
`In[]: calcPoints('iou')` `Out[]:` *Show what is returned:* 17	*Show what is printed:* i 1 o 7 u 17
`In[]: calcPoints('R45ot')` `Out[]:` *Show what is returned:* 7	*Show what is printed:* R 3

Understanding for loop 2: Conditionals in loops in `getScore`

This problem involves the following `getScore` function:

```
def getScore(word):
    score = 0
    for char in word:
        if char.isdigit():
            score = score + int(char)
            print(char, 'return1', score)
            return score
        elif char == 't':
            score = 10 # note: this uses =, not +=
        elif char in 'aeiou':
            score = score * 2
        else:
            score = score + 1
        print(char, 'if1', score)
        if char < 'i':  # compare by dictionary order
            if char > 'c':  # compare by dictionary order
                score += 5
            score += 3
        print(char, 'if2', score)
    print(char, 'return2', score)
    return score
```

For each of the following calls of `getScore`, show the output of all the print statements and what is returned.

Call	What is printed	What is returned

`getScore("show")`	`s if1 1`
	`s if2 1`
	`h if1 2`
	`h if2 10`
	`o if1 20`
	`o if2 20`
	`w if1 21`
	`w if2 21`
	`w return2 21`
		`21`

`getScore("pots")`	`p if1 1`
	`p if2 1`
	`o if1 2`
	`o if2 2`
	`t if1 10`
	`t if2 10`
	`s if1 11`
	`s if2 11`
	`s return2 11`
		`11`

`getScore("cat32")`	`c if1 1`
	`c if2 4`
	`a if1 8`
	`a if2 11`
	`t if1 10`
	`t if2 10`
	`3 return1 13`
		`13`

Understanding for loops 3: Tracing for loops and conditionals

Given the function `wordScore`, show what is (1) returned and (2) printed by the following function calls.
If nothing is printed, write "nothing."

The predicate `isVowel` returns `True` for vowels (any letter in the string "aAeEiIoOuU") and `False` otherwise.

```
def wordScore(word):
    result = 0
    if word[1] == word[0]:
        result += 1
    for char in word:
        if char in "!?":
            result += 100
            return result
        if isVowel(char):
            print('+',char)
            result += 2
        else:
            print('++', char)
    print('Finished!')
    return result
```

```
In[]: wordScore('et&c')
Out[]: Show what is returned:
2
```

Show what is printed:

```
+ e
++ t
++ &
++ c
Finished!
```

```
In[]: wordScore('eek')
Out[]: Show what is returned:
5
```

Show what is printed:

```
+ e
+ e
++ k
Finished!
```

```
In[]: wordScore('z?y!')
Out[]: Show what is returned:
100
```

Show what is printed:

```
++ z
```

Understanding for loops 4: Tracing for loops and conditionals

You are given the function `string_inspector` that contains conditional statements. Trace the execution of invoking this function with different arguments by showing what is **printed** and what is **returned** for each invocation.

- `char.upper()` returns the uppercase version of `char` if it's a letter; otherwise it just returns `char`.
- `char.isupper()` returns `True` if `char` is an uppercase letter and `False` otherwise

```
def string_inspector(s):
    result = '$'
    for char in s:
        result += char.upper()

        if char.isupper():
            print("P")

        if char == 'a':
            if 'd' in s:
                print("Q")
            else:
                print("R")
        elif char == 'b':
            if s[-1] == 'd':
                print("S")
            else:
                print("T")
        elif char in 'cdef':
            print("U")
            if s[0] == 'z':
                return result
        else:
            print("W")

    if len(s) >= 4:
        return result + "!"
    else:
        return result + "*"
```

```
>>> string_inspector("Abba")
```

Show what is **printed**:	Show what is **returned**:
P W T T R	'$ABBA!'

```
>>> string_inspector("zemor")
```

Show what is **printed**:	Show what is **returned**:
W U	'$ZE'

```
>>> string_inspector("bad")
```

Show what is **printed**:	Show what is **returned**:
S Q U	'$BAD*'

Understanding for Loops 5: Tracking variables

This problem involves the following function definition that uses the **tracking variable** `prev` to keep track of the previous letter in the word while the **`for`** loop is executed. You may assume that `isVowel` correctly returns `True` if its string argument is a lower or upper case version of the letters `a`, `e`, `o`, `i`, `u`, and is otherwise `False`.

```
def process(word):   # line 1
    newWord = ''         # line 2
    prev = ''            # line 3
    for letter in word:  # line 4
        if isVowel(letter) or isVowel(prev):  # line 5
            newWord += letter                 # line 6
        # print('prev', prev, 'letter', letter, 'newWord', newWord) # line 7
        prev = letter    # line 8
    return newWord       # line 9
```

Part a: [6 pts] Suppose the debugging `print` on line 7 is uncommented. Fill in the underlined parts in the following printed output to show what is printed when `process('purple')` is called. if the empty string is printed, leave the underlined part blank.

```
prev ____ letter p___ newWord _____________
prev p___ letter u___ newWord u____________
prev u___ letter r___ newWord ur___________
prev r___ letter p___ newWord ur___________
prev p___ letter l___ newWord ur___________
prev l___ letter e___ newWord ure__________
```

Part b: [3 pts] Show the **result** returned by the following three calls to `process`. Assume that line 7 is commented out, so that nothing is printed. Write the result value after the arrow ⇒, remembering to quote all string values.

`process('length')` ⇒ `'en'` `process('odious')` ⇒ `'odious'` `process('bcd')` ⇒ `''`

Understanding for loops 6: Debugging a loop

This problem involves a function `hasThreeConsecutiveVowels` that should return `True` when called on a string that contains at least three consecutive vowels and `False` for any other string. For example, it should return `True` for strings like `"bureau"`, `"precious"` , and `"queue"` and `False` for strings like `"cat"`, `"nation"`, and `"evoke"`.

Below is a buggy version of `hasThreeConsecutiveVowels` that does not work correctly.

```
def buggyHasThreeConsecutiveVowels(string):  # line 1
    counter = 0                              # line 2
    for letter in string:                    # line 3
        if isVowel(letter):                  # line 4
            counter += 1                     # line 5
            if counter == 3:                 # line 6
                return True                  # line 7
    return False                             # line 8
```

Assume that the function `isVowel` correctly returns `True` when its single argument is a vowel (a single letter in `aeiouAEIOU`) and `False` otherwise.

Part 3a Are there any counterexample strings for which `buggyHasThreeConsecutiveVowels` returns `True` when `hasThreeConsecutiveVowels` returns `False`?

- If yes, give an example of such a counterexample string, and explain in English the structure of such counterexample strings.
- If no, explain why such counterexample strings are not possible.=

Yes. buggyHasThreeConsecutiveVowels returns True for **any** string that contains at least three vowels, even when there are not three **consecutive** vowels. E.g. `'nation'`, `'abei'`, `'hahaha'`, `'soup du jour'`

Part 3b Are there any counterexample strings for which `buggyHasThreeConsecutiveVowels` returns `False` when `hasThreeConsecutiveVowels` returns `True`?

- If yes, give an example of such a counterexample string, and explain in English the structure of such counterexample strings.
- If no, explain why such counterexample strings are not possible.

No. buggyHasThreeConsecutiveVowels returns True for **every** string that contains at least three vowels, so it will never return False for **any** string that has three consecutive vowels.

Part 3c It is possible to add code between two consecutive lines of `buggyHasThreeConsecutiveVowels` so that the modified function behaves like the correct `hasThreeConsecutiveVowels`. Specify the two lines between which the new code should be added and what the new code is.

Between lines 7 and 8, add the code:

```
        else: # matches the outer if, not the inner if
            counter = 0
```

Defining functions with loops

Defining functions with loops 1: Hiding characters

Define a function named `hide` that takes a string and replaces certain characters with a `'*'`. The `hide` function will take two parameters: (1) a string and (2) a string of characters such that if any of them occur in the first string parameter, they are to be hidden (replaced) by a `'*'`. For full credit, `hide` should contain one `for` loop.
Below are some sample invocations.

Invocation	Result
`hide('apple', 'p')`	`'a**le'`
`hide('apple', 'pa')`	`'***le'`
`hide('coffee', 'oe')`	`'c*ff**'`
`hide('coffee', 'xyz')`	`'coffee'`
`hide('winter is coming', 'coming')`	`'w**ter *s ******'`

Define your `hide` function in the box below:

```
def hide(string, charsToHide):
    result = ''
    for char in string:
        if char in charsToHide:
            result += '*'
        else:
            result += char
    return result
```

Defining functions with loops 2: Duplicating odd characters

Define a function `duplicateOddChars` that takes a string as its single argument and returns a string containing all the characters of the given string in order **except** that each character at an **odd index** is duplicated.

- Recall that indexing starts at 0.
- For **full credit**, your function **must** use a **while** loop (but **substantial partial credit** will be awarded if it uses a correct **for** loop).

Below are shown some examples of invoking the function.

```
>>> duplicateOddChars('Omaha, NE')
'Ommahha,, NNE'

>>> duplicateOddChars('ba')
'baa'

>>> duplicateOddChars('yes!')
'yees!!'
```

```
>>> duplicateOddChars('I')
'I'

>>> duplicateOddChars('')
''
```

Define your `duplicateOddChars` function in this box.

```
# while loop solution
def duplicateOddChars(word):
    result = ''
    index = 0
    while index < len(word):
        if index % 2 == 1:
            result += word[index] * 2
        else:
            result += word[index]
        index += 1
    return result

# for loop solution
def duplicateOddChars(word):
    result = ''
    for index in range(len(word)):
        if index % 2 == 1:
            result += word[index] * 2
        else:
            result += word[index]
    return result
```

Defining functions with loops 3: Shouting a string

Define a function `shout` that takes a string as its single argument and **returns** a version of the string in which

- all alphabetic characters **and spaces** are kept but all other non-space non-alphabetic characters have been removed;
- all alphabetic characters have been capitalized.

Below are some sample invocations of `shout`:

```
>>> shout('{one}, (two), [three]')
'ONE TWO THREE'

>>> shout('!@Foo#$BAR%^baz&*')
'FOOBARBAZ'
```

```
>>> shout('You say "Goodbye!", and I say "Hello!"')
'YOU SAY GOODBYE AND I SAY HELLO'

>>> shout('')
''
```

- In your `shout` definition, you may use either a **`for`** loop or a **`while`** loop, whichever you find easier.
- If s is a string, then `s.isalpha()` returns `True` if all the characters in s are alphabetic, and False otherwise.
- You can test for a space character using `==`.
- If s is a string, then `s.upper()` returns a version of s in which all alphabetic characters are capitalized.

Define your **`shout`** *function in this box.* ***Make sure that your indentation is clear!***

```
# while loop version
def shout(text):
    result = ''
    index = 0
    while index < len(text):
        char = text[index]
        if char == ' ' or char.isalpha():
            result += char.upper()
        index += 1
    return result

# for loop version
def shout(text):
    result = ''
    for char in text:
        if char == ' ' or char.isalpha():
            result += char.upper()
    return result
```

Defining functions with loops 4: Swapping case in a string

Define a function `swapCase` that takes a string as its single argument and **returns** a version of the string in which

- all alphabetic characters **and spaces** are kept, but all other non-space non-alphabetic characters have been removed;
- all lower-case alphabetic characters have been upper-cased and all upper-cased alphabetic characters have been lower-cased. (An upper-case letter is just a capital letter; a lower-case letter is not a capital.)

Below are some sample invocations of `swapCase`:

```
>>> swapCase('lower Capitalized UPPER')
'LOWER cAPITALIZED upper'

>>> swapCase('I am NOT shouting but TALKING!')
'i AM not SHOUTING BUT talking'

>>> swapCase('Happy Birthday! You're the *BEST*') =>
'hAPPY bIRTHDAY yOURE THE best'
```

- In your `swapCase` definition, you should use a **`for`** loop to get full credit
- If s is a string, then
  - `s.isalpha()` returns `True` if **all** the characters in s are alphabetic, and `False` otherwise.
  - `s.islower()` returns `True` if **all** the alphabetic characters in s are lower case, and `False` otherwise. (`s.isupper()` is similar for upper case , but it's not necessary in this problem.)
  - `s.lower()` returns a version of s in which all letters are lower-cased.
  - `s.upper()` returns a version of s in which all letters are upper-cased.
- You can test for a space character using `==`.

Define your `swapCase` *function in this box.* ***Make sure that your indentation is clear!***

```
def swapCase(string):
    result = ''
    for char in string:
        if char.isalpha():
            if char.islower():
                result += char.upper()
            else:
                result += char.lower()
        elif char == ' ':
            result += char
    return result
```

Defining functions with loops 5: Laughing strings

Part 8a: Define a function named `laughStretch` with four parameters — two strings (a word and a filler character) and two integers (maxReps and length) — that **prints** stretched words like the ones shown below. The function repeats the word until it reaches the target length or runs out of allowed repetitions, and then fills any leftover space with the filler character.

- `maxReps` is the maximum number of times that word can be repeated in the final string
- `length` is the minimum length of the final string
- You can assume that the `filler` argument will contain a single character.
- Recall that * can be used to repeat a string.
- For **full credit**, your function **must** use a **`while`** loop (but **substantial partial credit** will be awarded if it uses a correct **`for`** loop).

Below are shown some examples of invoking the function.

```
>>> laughStretch("ha","~",10,6)
hahaha

>>> laughStretch("ha","~",2,6)
haha~~

>>> laughStretch("teehee","!",1,10)
teehee!!!!
```

```
>>> laughStretch("ha","w",2,7)
hahawww

>>> laughStretch("ha","w",4,7)
hahahaha
```

Define your `laughStretch` *function in this box.*

```
# while loop solution
def laughStretchWhile(word, fillerChar, maxReps, targetLength):
    resultSoFar = ''
    while (len(resultSoFar) < targetLength) and maxReps > 0:
        resultSoFar += word
        maxReps -= 1
    charsNeeded = targetLength-len(resultSoFar)
    if charsNeeded > 0:
        resultSoFar += fillerChar*charsNeeded
    print resultSoFar

# for loop solution
def laughStretchFor(word, fillerChar, maxReps, targetLength):
    resultSoFar = ''
    for _ in range(maxReps):
        resultSoFar += word
        if len(resultSoFar) >= targetLength:
            break
    charsNeeded = targetLength-len(resultSoFar)
    if charsNeeded > 0:
        resultSoFar += fillerChar*charsNeeded
   print resultSoFarv
```

Part 8b: (You do not have to define `laughStretch` correctly in part 4a in order to answer this part.) In the box below, write an invocation of the `laughStretch` function that will display the pattern below if `laughStretch` is correctly defined:

```
teehee!teehee!!!
```

Write your invocation of `laughStretch` *in this box:*

```
laughStretch('teehee!', '!', 2, 16)
```

Defining functions with loops 6: Converting a tracking variable to an index loop

This problem is related to the one in Understanding for Loops 5: Tracking variables. It involves the following function definition that uses the **tracking variable** `prev` to keep track of the previous letter in the word while the **`for`** loop is executed. You may assume that `isVowel` correctly returns `True` if its string argument is a lower or upper case version of the letters `a`, `e`, `o`, `i`, `u`, and is otherwise `False`.

```
def process(word):   # line 1
    newWord = ''         # line 2
    prev = ''            # line 3
    for letter in word:  # line 4
        if isVowel(letter) or isVowel(prev):  # line 5
            newWord += letter                 # line 6
        # print('prev', prev, 'letter', letter, 'newWord', newWord) # line 7
        prev = letter    # line 8
    return newWord       # line 9
```

Below, show how to define an alternative version of `process` that uses an **index loop** expressed with a **`while`** loop rather than a tracking variable expressed with a **`for`** loop to handle accessing the previous letter. The definition has been started for you; you must complete it. You should ***not*** include the commented debugging print from line 7 in your definition.

```
def process(word):
    """version of process with an index loop using while"""
    index = 0
    newWord = ''
    while index < len(word)     :
        letter = word[index]
        if isVowel(letter) or (index !=0 and isVowel(word[index-1])):
            newWord += letter
    return newWord
```

Defining functions with loops 7: `replaceVowelSequences` *[tests writing a complex loop]*

Because vowels are more likely to change over time than consonants, linguists sometimes describe words in terms of just their consonants, putting in a star (asterisk) for a sequence of consecutive vowels. So `'dog'` would be written `'d*g'` and `'seafood'` would be written as `'s*f*d'`.

In this problem you will write a function `replaceVowelSequences` that takes a word and returns a string that replaces each **sequence** of vowels in the word with a single asterisk. Here are iteration tables that show the function working on some examples:

Iteration Tables

Example 1: `'dog'`

char	result	inVowelSequence
	''	False
d	'd'	False
o	'd*'	True
g	'd*g'	False

Example 2: `'seafood'`

char	result	inVowelSequence
	''	False
s	's'	False
e	's*'	True
a	's*'	True
f	's*f'	False
o	's*f*'	True
o	's*f*'	True
d	's*f*d'	False

Define `replaceVowelSequences` in the box below using a **for** loop with the state variables shown in the above iteration tables. Assume there is a correct `isVowel` predicate that you can use without defining it.

```
# SOLUTION 1: A completely correct solution has the following properties:
#   * It uses a for loop with three state variables having exactly the names
#       char, result, and inVowelSequence, as shown in the iteration tables.
#  *  The code in the for loop body expresses the update rules for the result
#  result and inVowelSequence state variables that are implied by the iteration tables.
def replaceVowelSequences(word):
    result = '' # initialize state variable for accumulating result string
    inVowelSequence = False # initialize state variable that determines when to add '*'
    for char in word: # iterate over each character in word, using char as iteration variable
        if isVowel(char):
            if not inVowelSequence: # add '*' only when previous char was not vowel
                result += '*'
                inVowelSequence = True # For next time, indicate previous char *was* a vowel
        else: # use else rather than testing `not isVowel(char)`
            result += char # always add a nonvowel to result
            inVowelSequence = False # For next time, indicate previous char was *not* a vowel
    return result
```

```
# SOLUTION 2:Observe that inVowelSequence needn't actually be a state variable, since the value
# of inVowelSequence is the result of the expression result != '' and result[-1] == '*'
# Based on this observation, a simplified version of the function definition is:
def replaceVowelSequences(word):
    result = '' # initialize state variable for accumulating result string
    for char in word: # iterate over each character in word, using char as iteration variable
        if isVowel(char):
            if not (result != '' and result[-1] == '*') # add '*' only when previous char was not
vowel
                # if can be simplified to if (result == '' or result[-1] != '*')
                result += '*'
        else: # use else rather than testing `not isVowel(char)`
            result += char # always add a nonvowel to result
    return result
```

Defining functions with loops 8: firstDigits *[tests writing a complex loop]*

Define a function named `firstDigits` that takes a string containing only spaces and digits and returns a string containing the first digits from each group of digits. `firstDigits` has a single string parameter and returns a string.

If the string passed into `firstDigits` is not empty, it will always begin with a digit and end with a digit. If the string is empty, the function should return the string "NOTHING!" You can assume the groups of digits are separated by single spaces.

For full credit, `firstDigits` must contain exactly one while loop or for loop, and cannot use .split().
Here are some sample function calls:

Invocation	Result
`firstDigits('19 500 0')`	`'150'`
`firstDigits('')`	`'NOTHING!'`
`firstDigits('34 34 34 34')`	`'3333'`
`firstDigits('1 2 3')`	`'123'`

Write your `firstDigits` function in the box below:

```
def firstDigits1(string):
    ''' Version of firstDigits with value loop and tracking variable. '''
    if string == '':
        return 'NOTHING!'
    prevSpace = True
    digits = ''
    for char in string:
        if char == ' ':
            prevSpace = True;
        else:
            if prevSpace:
                digits += char
            prevSpace = False;
    return digits

def firstDigits2(string):
    ''' Version of firstDigits with while-based index loop. '''
    if string == '':
        return 'NOTHING!'
    digits = string[0] # guaranteed to be a digit
    index = 0
    while index < len(string) # don't process last index of string!
        if string[index] == ' ' and string[index+1] != ' ':
            # By assumption, string guaranteed not to *end* in a space,
            # so string[index+1] will never be out-of-bounds.
            # Alternatively, continuation condition can be: index < len(string)-1
            digits +=  string[index+1]
    return digits
```