

Math 158 Python crash course

Nathan Pflueger

14 September 2015

This document provides a series of worked examples showing many of the basic notions from the Python programming language that we will need for Math 158. The guiding goal is to work through the four problems posted on the following demonstration problem set, but I will occasionally go on short tangents to show some other features of the language.

<https://www.hackerrank.com/math158-demonstration-pset>

Before working through this document, you should install Python on your machine if it is not already present. You can find installation instructions at the beginning of the following Python tutorial. It will likely also be useful to begin working through the exercises (at least exercises 0 through 4). The entire tutorial is excellent, so you may wish to use it instead of, or in addition to, the examples I give here.

<http://learnpythonthehardway.org/book/>

This document is certainly not comprehensive. We will continue to encounter new language features that will be useful for our purposes as the semester goes on, and you should also search online, or talk to other programmers you know, to teach yourself any other features or techniques that look useful or interesting.

General suggestions

- Like anything else, you can only learn programming by doing it. Don't read these examples; actually type them in on your machine and see what happens.
- Experiment constantly. Try changing what I've written, and see how the output changes. Invent new examples, and try to predict in advance what will happen when you run them. See if you can write the same code several different ways. See if you can improve on my code to make it more readable, more useable, faster, etc.
- Use a search engine liberally. If you're getting an error, find someone else that's encountered it (almost certainly someone has). If you can't figure out how to do a task, or you suspect there's a cleaner way to do it, search for it.
- Read other people's code whenever possible, and learn from it. Find code on help message boards, in Python tutorials and documentation, etc.

Organization of this document

Each numbered section is named after either one of the programming tasks from the sample assignment, or a basic feature of the Python language. These features appear in the order that we encounter them while solving the problems on the sample assignment.

Throughout, any text in **typewriter font** is either code or the output of that code. Lines that begin `>>>` indicate where I have typed input into the python terminal, while the lines that immediately follow are the output. When code is enclosed in a box, that indicates that I have saved this code to a file.

Contents

1	Variables, printing, basic arithmetic	2
2	Problem 1	3
3	Source files	5
4	Problem 2 statement	5
5	Lists	6
6	for loops	8
7	Fuctions and def	9
8	Solving problem 2	10
9	Problem 3 statement	10
10	if and else	11
11	while loops	12
12	Problem 3 solution	13
13	Building up lists	14
14	Problem 4 statement	15
15	Converting characters to integers	16
16	Rotating entire phrases	17
17	elif	19
18	Problem 4 solution	19

1 Variables, printing, basic arithmetic

Let's start by firing up the Python terminal. You should get a couple lines of boilerplate, like the following.

```
Python 2.7.10 (default, Jun 10 2015, 19:43:32)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
```

If the first line says “Python 3.xxx” instead of “Python 2.xxx”, you are running Python 3, which is a different language. The differences aren't huge, but they could become confusing. I suggest installing Python 2 and using it instead.

To begin, let's define two variables x and y , print their values, and do some arithmetic with them.

```
>>> x = 5
>>> y = 7
>>> print x
5
```

```
>>> print y
7
>>> print x+y
12
>>> print x,y,x+y
5 7 12
```

The last line shows an often-convenient feature: to print several space-separated numbers (or other data), type “print” followed by these variables, separated by commas.

What happens if we try to print a variable that doesn’t exist yet?

```
>>> print z
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'z' is not defined
```

This is the first error message we’ve seen. Error messages are sometimes easy to understand, and sometimes very confusing. In this case, there’s a little of both. The first two lines look obscure, but the last one tells you your problem pretty clearly.

We can rectify this situation by defining z , and telling the interpreter its value. Then we can print it.

```
>>> z = x*y
>>> print z
35
```

2 Problem 1

Take a look at the first problem in the demonstration assignment.

Problem Statement

You are given an integer n . You must print $n + 1$.

Input Format

An integer n .

Constraint:

$1 \leq n \leq 100$

Output Format

The integer $n + 1$.

You’ll notice that there’s a bit of starter code in the submission window already, which reads an integer n from standard input. All we need to do is compute $n + 1$ and print it. This is accomplished with the single line `print n+1` at the end.

Current Buffer (saved locally, editable) Python 2

```

3 # Read the number n
4 n = int(raw_input())
5
6 # Your code here
7 print n+1

```

Line: 1 Col: 1

Test against custom input

There are two submission buttons. The first one, “run code”, will test your code by showing you what it does on a few provided sample cases.

Testcase 0 ✓ Testcase 1 ✓ Testcase 2 ✓

Congratulations, you passed this test case!

Input (stdin)

64

Your Output (stdout)

65

Expected Output

65

In this case, we see that the code passes the three sample cases. If it didn’t we could compare our program’s output to the desired output and attempt to diagnose the problem. Once you’re ready to run your code on the full battery of tests, hit the “submit code” button. You’ll see a screen like the following.

Submitted a few seconds ago • Score: 10.00 Status: Accepted

Test Case #0: ✓ 0.01s	Test Case #7: ✓ 0.01s	Test Case #14: ✓ 0s
Test Case #1: ✓ 0.01s	Test Case #8: ✓ 0s	Test Case #15: ✓ 0.01s
Test Case #2: ✓ 0s	Test Case #9: ✓ 0.01s	Test Case #16: ✓ 0.01s
Test Case #3: ✓ 0.01s	Test Case #10: ✓ 0s	Test Case #17: ✓ 0s
Test Case #4: ✓ 0.01s	Test Case #11: ✓ 0.01s	Test Case #18: ✓ 0.01s
Test Case #5: ✓ 0.01s	Test Case #12: ✓ 0s	Test Case #19: ✓ 0s
Test Case #6: ✓ 0.01s	Test Case #13: ✓ 0s	

The code is now run on 20 different tests, and a score is computed based on how many of them the code solves correctly. Unlike the sample cases, you are not allowed to view the input and output of these tests until after the assignment deadline. Once the assignment deadline has passed, a button will appear when you mouse-over a test case to download the input and output, so you can see which inputs caused problems for your code.

3 Source files

The first problem was simple enough that it was easy to type the solution right into the window. For other problems, you'll want to write out a more detailed program on your own machine before testing it on hackerrank.

In principle, you can type all of your code directly into the Python terminal. In practice, however, it's better to type your longer definitions and blocks of code into separate files. For example, I could write the following text to the file `converstions.py`.

```
minutes_per_hour = 60
hours_per_day = 24
days_per_year = 365

#From these, compute the minutes in a year
minutes_per_year = minutes_per_hour * hours_per_day * days_per_year
```

If I now type `execfile('conversions.py')` in the Python interpreter, it will run all the code in the file as if I typed it in directly. All the variables I defined in the file will now be accessible in the interpreter.

```
>>> execfile('conversions.py')
>>> print minutes_per_year
525600
```

One advantage of editing and executing source files is that it is easier to fix mistakes. For example, if I want the *average* number of days per year, not the number of days per year in a non-leap year, I can go make the following change to the source file,

```
minutes_per_hour = 60
hours_per_day = 24
days_per_year = 365.25

#From these, compute the minutes in a year
minutes_per_year = minutes_per_hour * hours_per_day * days_per_year
```

and then instantly bring everything up to date.

```
>>> execfile('conversions.py')
>>> print minutes_per_year
525960.0
```

Another advantage of using source files is that I can save them for use in multiple sessions, rather than typing all my code in from scratch each time. It will also make it easy to instantly submit code to hackerrank.

4 Problem 2 statement

Before going on, take a look at the problem statement for the next problem on the demonstration set.

Problem Statement

You are given a space-separated list of integers. You must print the product of all the integers in this list.

The first ten test cases have exactly two numbers in the list. The remaining ten test cases may have any number of numbers in the list (possibly only one).

Input Format

A space-separated list of integers.

Constraints:

Each integer is between 1 and 9 inclusive.

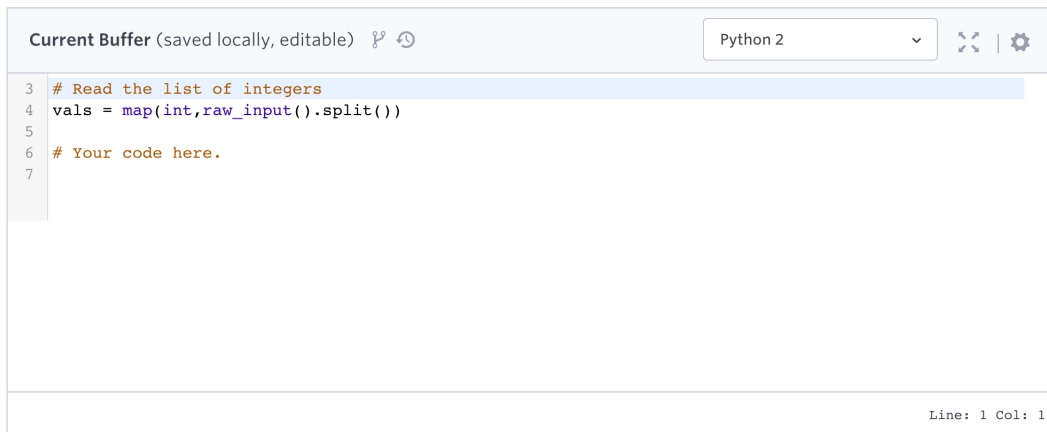
First 10 test cases: the list has two integers.

Last 10 test cases: the list has between 1 and 9 integers, inclusive.

Output Format

An integer.

The problem also comes with this starter code.



```
Current Buffer (saved locally, editable) Python 2
3 # Read the list of integers
4 vals = map(int,raw_input().split())
5
6 # Your code here.
7
Line: 1 Col: 1
```

For this problem, we must use more than ordinary variables. The reason is that the program needs to be able to multiply two numbers, or three numbers, and so on, without knowing in advance how many numbers there will be. The new type of variable that we must use is a *list*.

5 Lists

That first line in the starter code will process a sequence of numbers (separated by spaces) and write something called a list to “val.” Here’s what happens if you type it into the Python interpreter and print the result.

```
>>> vals = map(int, raw_input().split())
2 4 6 0 1
>>> print vals
[2, 4, 6, 0, 1]
```

The variables “vals” is a list of length 5. When you print a list, these values are printing inside square brackets, separated by commas. You can also use the same syntax to declare your own lists. Lists don’t

have to consist of integers – one of the lists below is a list of strings instead of integers (you can also mix data types within a single list if you wish).

```
>>> odds = [1,3,5]
>>> troch = ['double', 'double', 'toil', 'trouble']
>>> print odds
[1, 3, 5]
>>> print troch
['double', 'double', 'toil', 'trouble']
```

You can extract the individual elements of a list using square brackets as shown below. You will see an error if you ask for an element that isn't there.

```
>>> print troch[0]
double
>>> print troch[1]
double
>>> print troch[2]
toil
>>> print troch[3]
trouble
>>> print troch[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Less important note: One unusual feature of Python (that could ruin your day if you tried to do it in C++) is that you actually can ask for list elements with negative indices. If you use negative indices, you will count from the *back* of the list instead of the front. For example:

```
>>> print troch[-1]
trouble
```

You can generate a list of consecutive integers in a single command using **range** as follows.

```
>>> counting = range(100)
>>> print counting
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24,
25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49,
50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74,
75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99]
```

An seemingly odd thing about *range* is that the list you get doesn't include the number you wrote in (here, 100). The reason is that the 100 tells it the number of elements you want in the list, and that 0 is in the list. This seems odd at first, but it turns out to be very convenient in many purposes, as you'll see over time. For now, just remember it.

If you have a long list, it's often handy to be able to look up features of it, like its sum or its length, quickly. There are some built-in commands for this kind of thing. You can find many more with a little time in a search engine.

```
>>> sum(counting)
4950
>>> len(counting)
100
```

By the way, `range` can also give you lists of consecutive integers that start where you like. The rule is that the list always starts with the first number you give, and goes up to *but not including* the second number. Again, this seems odd at first, but it has nice features. For example, the list `range(a,b)` has exactly $b - a$ elements in it, not $b - a + 1$.

```
>>> print range(10,20)
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
>>> len( range(10,20) )
10
```

6 for loops

For the problem we're trying to solve on hackerrank, we have a list of numbers, and we want them all multiplied together. So we'll want to process these numbers one by one, and do something (namely, multiply) with each one. The standard programming concept to do this (look at each element in a list and "do something" with it) is the "for" loop. Here is a basic example, using the list `troch` that we defined earlier (the name is short for "trochaic tetrameter," a somewhat unusual meter that famously occurs in MacBeth in the scene that this list makes reference to).

```
>>> for word in troch:
...     print word
...
double
double
toil
trouble
```

The keyword `for` means that we're going to examine each element in a list. The last word tells the interpreter which list. The *word* tells the interpreter to make each of these elements available, in turn, as a variable called `word`. Then I've typed a colon, and put some other code below, indented. All subsequent indented lines will become part of the loop.

The variable `word` is only defined inside the loop. If you try to ask the interpreter about `word` after the loop has finished, you will find that it no longer exists. It plays a role similar to *dummy variables* in calculus (indeed, you can think of an integral as a kind of for loop if you like).

By the way, if I wanted to print all these list elements on one line instead of each on their own line, I could accomplish this by simply putting a comma at the end of `print word` as follows.

```
>>> for word in troch:
...     print word,
...
double double toil trouble
```

If I like, I can also compress the code a bit if my for loop only has one command in it (as it does here) by just putting the code right after the colon, as follows.

```
>>> for word in troch: print word,
...
double double toil trouble
```

That's enough generalities about *for* loops. Now let's use them to solve the problem at hand: multiplying all the numbers in a list together. Here the "do something" that we want to do with each element in our list is multiply our result by it. Here's some code to achieve this.


```

>>> prod = 1
>>> for n in [1,2,3,4,5]:
...     prod = prod * n
...
>>> print prod
120

```

I mentioned earlier that n is a dummy variable here. Indeed, I could replace n with any variable name I like, and the code will do exactly the same thing.

```

>>> prod = 1
>>> for cthulu in [1,2,3,4,5]:
...     prod = prod * cthulu
...
>>> print prod
120

```

7 Functions and def

That for loop that I used to multiply the elements in a list together is useful enough that I'd like to be able to use it multiple times without typing it out from scratch each time. One nice way to do this is to wrap it up in a *function*. The following code does this.

```

>>> def mult_list(ls):
...     prod = 1
...     for elt in ls:
...         prod *= elt # This is shorthand for prod = prod * elt
...     return prod

```

The `def` keyword tells the interpreter that I'm defining a new function. The `ls` is called an argument – like the dummy variables used in a `for` loop, it has no existence outside of the function definition. Its job is to give a name to some data that the user will supply whenever they call this function.

You can now call this function using the following syntax. Note that you either type in your own list from scratch, or you can give the name of a variable that you've already defined elsewhere. That variable is called an *argument*; it will get named `ls` and used in the function definition code.

```

>>> mult_list([1,2,3,4,5])
120
>>> mult_list(odds)
15

```

Note that you'll get an error if you give your function an argument that it doesn't know how to multiply the elements of. For example, the elements of the list `troch` are strings, not integers, so the interpreter doesn't know how to multiply them.

```

>>> mult_list(troch)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in mult_list
TypeError: can't multiply sequence by non-int of type 'str'

```

8 Problem 2 solution

To solve problem 2, let's first of all write that function definition to a file for later use. The following will now be `mult.py`. Whenever we execute it, we'll have access to this function.

```
def mult_list(ls):
    prod = 1
    for elt in ls:
        prod *= elt
    return prod
```

```
>>> execfile('mult.py')
```

Now, to solve the problem on hackerrank, we can just add the line from the starter code that reads a list from standard input, and finally add one line to print the product of those numbers (computed with `mult_list`) to standard output.

```
def mult_list(ls):
    prod = 1
    for elt in ls:
        prod *= elt
    return prod

vals = map(int,raw_input().split())
print mult_list(vals)
```

Before uploading, we can quickly test our program to make sure it acts how we expect.

```
>>> execfile('mult.py')
2 4 3
24
```

The second line, `2 4 3`, are the numbers I typed in (on hackerrank, the platform will feed these in). The `24` is the output. It looks like the program is running as expected, so let's submit it.

Rather than typing the code in by hand, you can use this button to upload `mult.py` directly.

[📁 Upload Code as File](#)

Now submit the code as before. It should pass all test cases.

9 Problem 3 statement

Here's the statement of the third problem on the demonstration set, along with a sample.

You are given a positive integer n . You should repeatedly perform the following procedure on n .

- If n is equal to 1, stop.
- If n is even, replace n with $\frac{1}{2}n$.
- If n is odd, replace n with $3n + 1$.

Your program should print each value of n that occurs as you repeat this procedure (including the original value and 1).

It is a well-known open problem (the Collatz Conjecture) whether this procedure will always terminate regardless of n . Suffice it to say that all of the given inputs will give a procedure that terminates.

Input Format

A positive integer n .

Constraints:

$2 \leq n < 200$

The described procedure is guaranteed to terminate on n .

Output Format

A sequence of positive integers, one on each line. The first line should contain the original integer, and the last line should contain "1".

Sample Input

6

Sample Output

6
3
10
5
16
8
4
2
1

The main new language element that we need to deal with these is a way to distinguish between two cases (here, that n is even or that n is odd) and execute different code in these two cases.

10 if and else

This need to met by `if` and `else`. These keywords use similar syntax to the keywords `for` and `def`. A simple example is shown below. Here, suppose that we have first set the variable n to be equal to 5.

```
>>> if n%2 == 0:
...     print n/2
... else:
...     print 3*n+1
...
16
```

Since 5 is odd, the expression `n%2` evaluates to 1, else the code that was executed was `print 3*n+1`. Since we'll need to use this block of logic over and over again, let's write a function around it and write it to a file. I've written the following to `collatz.py`.

```
def flutter(n):
    if n%2 == 0:
        return n/2
    else:
        return 3*n+1
```

Upon executing this file, we can verify that the function is doing what we expect, namely telling the next value of n given its current value.

```
>>> execfile('collatz.py')
>>> flutter(5)
16
>>> flutter(6)
```

```
3
>>> flutter(7)
22
>>> flutter(8)
4
```

So far so good. What we really want to do, though, is apply this function over and over again. We need a type of loop to do this. The `for` loop already discussed is not well-suited to this purpose. The reason is that we don't know in advance how many elements we'll be printing out before we reach 1. Instead we introduce the other main type of loop.

Unimportant tangent: You actually can use a `for` loop to solve this problem if you use something called a *generator* in place of a list. Generators are a wonderful aspect of Python that often come in quite handy, but I won't go into them here to keep things brief.

11 while loops

Recall that a `for` loop executes a certain block of code repeatedly, once for each element in a list. A while loop does not walk through a list; instead, it continues to repeat until a certain condition fails to be met. The syntax is to write `while (condition):` following by an indented block. Here is an example.

```
>>> n = 5
>>> while n != 1:
...     n = flutter(n)
...     print n,
...
16 8 4 2 1
```

Here, n begins at 5 then it is repeatedly replaced with the next step in the Collatz sequence and printed on a single line.

We'll need to use this code in our solution, so I'll write it into `collatz.py`. I've also removed the comma at the end of the `print` line, since the final submission should print each value of n on a new line. This code actually has a problem with it that we'll see when it is submitted – see if you can spot it (I'll point it out and fix it in a moment).

```
def flutter(n):
    if n%2 == 0:
        return n/2
    else:
        return 3*n+1

def flight(n):
    while n != 1:
        n = flutter(n)
        print n
```

Here's a quick test to make sure the code does what it should.

```
>>> execfile('collatz.py')
>>> flight(5)
16
8
4
```

2
1

There's a small problem: according to the problem statement, the program's output should begin by printing the original number n , before it flutters at all. Our code currently skips the original value. This is easy to fix, though: just ask the function `flight` to first print the original number. Here's the new `collatz.py`, and a quick test.

```
def flutter(n):
    if n%2 == 0:
        return n/2
    else:
        return 3*n+1

def flight(n):
    print n
    while n != 1:
        n = flutter(n)
    print n
```

```
>>> execfile('collatz.py')
>>> flight(5)
5
16
8
4
2
1
```

12 Problem 3 solution

It looks like the `flight` function works. we can now add the input/output code to the end of `collatz.py` and submit it. It should pass all test cases.

```
def flutter(n):
    if n%2 == 0:
        return n/2
    else:
        return 3*n+1

def flight(n):
    print n
    while n != 1:
        n = flutter(n)
    print n

n = int(raw_input())
flight(n)
```

13 Building up lists

Before solving the last demonstration problem, I will point out how you can build a list up one element at a time. The operator `+` (which adds numbers) can also be used to append lists to each other. Here is an example.

```
>>> part1 = [1,2,3]
>>> part2 = [4,5,6]
>>> whole = part1 + part2
>>> print whole
[1, 2, 3, 4, 5, 6]
```

For an application, I will modify the `flight` function from the last problem so that it returns a *list* of the numbers in the sequence, rather than printing them out to the screen.

```
def flutter(n):
    if n%2 == 0:
        return n/2
    else:
        return 3*n+1

def flight(n):
    ls = [n]
    while n != 1:
        n = flutter(n)
        ls += [n] #Append the new n to the flight list
    return ls
```

```
>>> execfile('collatz.py')
>>> flight(5)
[5, 16, 8, 4, 2, 1]
```

One handy thing about having this list in hand is that we can use it to rapidly answer other questions about the flight path without printing the whole thing to the screen. For example, we can easily measure how long each number takes to reach 1.

```
>>> len(flight(5))
6
>>> len(flight(17))
13
>>> len(flight(19))
21
```

We can also very quickly perform experiments like the following, which runs through the numbers 1 through 20 and tells you two facts about that number's flight path: the highest that it reaches, and how many steps it takes. I've used some string formatting syntax here that I haven't mentioned in this tutorial; it won't be crucial to us in this course, but you can learn about it, for example, in exercises 5 through 10 of *Learn Python the Hard Way*.

```
>>> for n in range(1,21):
...     fl = flight(n)
...     print 'The flight from %d reaches a high point of %d, but lands
after %d steps.' % (n,max(fl),len(fl))
```

The flight from 1 reaches a high point of 1, but lands after 1 steps.
The flight from 2 reaches a high point of 2, but lands after 2 steps.
The flight from 3 reaches a high point of 16, but lands after 8 steps.
The flight from 4 reaches a high point of 4, but lands after 3 steps.
The flight from 5 reaches a high point of 16, but lands after 6 steps.
The flight from 6 reaches a high point of 16, but lands after 9 steps.
The flight from 7 reaches a high point of 52, but lands after 17 steps.
The flight from 8 reaches a high point of 8, but lands after 4 steps.
The flight from 9 reaches a high point of 52, but lands after 20 steps.
The flight from 10 reaches a high point of 16, but lands after 7 steps.
The flight from 11 reaches a high point of 52, but lands after 15 steps.
The flight from 12 reaches a high point of 16, but lands after 10 steps.
The flight from 13 reaches a high point of 40, but lands after 10 steps.
The flight from 14 reaches a high point of 52, but lands after 18 steps.
The flight from 15 reaches a high point of 160, but lands after 18 steps.
The flight from 16 reaches a high point of 16, but lands after 5 steps.
The flight from 17 reaches a high point of 52, but lands after 13 steps.
The flight from 18 reaches a high point of 52, but lands after 21 steps.
The flight from 19 reaches a high point of 88, but lands after 21 steps.
The flight from 20 reaches a high point of 20, but lands after 8 steps.

As another quick experiment, here's a short bit of code that tells you the longest flight among all n from 1 to 9999.

```
>>> longest_flight = 0
>>> for n in range(1,10000):
...     cand = len(flight(n))
...     if (cand > longest_flight): longest_flight = cand
...
>>> longest_flight
262
```

The longest flight lasts 262 steps. Try to figure out how to modify this code so that it will also tell you *which* integer n produces this longest flight (the answer is 6171).

14 Problem 4 statement

For the last sample problem, we want to implement a special case of the Caesar cipher, called Rot13. This is often used to conceal spoilers; it has the nice feature that enciphering and deciphering are exactly the same algorithm. Here's the problem statement.

Problem Statement

You will be given a string on a single line. You should modify the string as follows, and print the result.

- Any letter between a and m inclusive should be advanced 13 places in the alphabet. The case should not be changed (e.g. if it was uppercase, it should stay uppercase).
- Any non-letter character should be left unchanged.

Another way to say this is that all letters should advance 13 places, where the letter z "advanced" back around to a. That is, each letter advanced 13 places modulo 26.

Input Format

A string on one line.

Output Format

A string on one line.

Here is a sample.

Sample Input

The rain in Spain falls mainly on the plain.

Sample Output

Gur enva va Fcnva snyyf znvayl ba gur cynva.

To solve this, we first need to talk about converting characters to integers and back.

15 Converting characters to integers

Each character (e.g. letter) corresponds to a particular number. The association of numbers to letters most often used is called ASCII. These numbers are used to encode characters in binary. You can access the numbers corresponding to a character using *ord*. Note that uppercase letters correspond to different ASCII numbers than their lowercase counterparts.

```
>>> ord('A')
65
>>> ord('a')
97
```

Non-letter characters also have numbers in ASCII.

```
>>> ord('@')
64
```

To go in reverse, from ASCII numbers to characters, use *chr*.

```
>>> chr(65)
'A'
>>> chr(75)
'K'
```

Using *ord*, *chr*, and arithmetic operations in tandem, you can do things like shift a character a specific number of places.


```
>>> chr(ord('L')+5)
'Q'
```

For the problem at hand, we want to shift characters by 13 places. I will write the following definition to `rot13.py`. So far, this will only work with uppercase letters, since I compute the letter's index by comparing with an uppercase A.

```
def rotate_char(ch): #Only does capital letters for now
    index = ord(ch)-ord('A')
    index += 13 #Advance the letter 13 places
    return chr(ord('A')+index)
```

```
>>> execfile('rot13.py')
>>> rotate_char('B')
'Q'
>>> rotate_char('C')
'P'
>>> rotate_char('D')
'Q'
```

This should be sufficient for a single letter (there is a bug, which we'll catch and fix soon).

16 Rotating entire phrases

Now, to apply rotation to multiple letters in sequence, we can use a `for` loop. So far, we've only used for loops to iterate through lists; it turns out that you can use a string a lot like a list of characters (and in particular, iterator through the characters with `for`). Therefore we can write the function `rotate_phrase` to the same file. It used a for loop to move through the input string, and successively rotates and appends each character to the output string, which is then returned.

```
def rotate_char(ch): #Only does capital letters for now
    index = ord(ch)-ord('A')
    index += 13 #Advance the letter 13 places
    return chr(ord('A')+index)

def rotate_phrase(phrase):
    res = '' # Initially, the empty string
    for ch in phrase: #Rotate the characters, one by one
        res += rotate_char(ch)
    return res
```

Here's what happens when we test this on a simple phrase (all in uppercase, since that's all we've accommodated so far).

```
>>> execfile('rot13.py')
>>> rotate_phrase('THERAININSPAIN')
'aUR_NV[V[']NV['
```

Oh no! We seem to have overlooked something. Some of these letters, like T, are going to non-letter characters. The reason is that they have been advanced 13 places, which sometimes pushes them past the end of the uppercase alphabet. We are *advancing* the characters, but we should be *rotating* them. We can fix this easily, though: instead of adding 13 to the index, we can add 13 and then take the remainder upon division by 26. Here is the modified file.

```

def rotate_char(ch): #Only does capital letters for now
    index = ord(ch)-ord('A')
    index = (index+13)%26 # **ROTATE** the letter 13 places
    return chr(ord('A')+index)

def rotate_phrase(phrase):
    res = '' # Initially, the empty string
    for ch in phrase: #Rotate the characters, one by one
        res += rotate_char(ch)
    return res

```

The function `rotate_phrase` now works properly for any string of uppercase letters.

```

>>> execfile('rot13.py')
>>> rotate_phrase('THERAININSPAIN')
'GURENVAVAFCNVA'

```

Before we can submit this code, we must generalize it two times. First, it must accommodate lowercase letters; second, it must accommodate non-letter characters.

To deal with lowercase letters, we can use the functions `isupper` and `islower`, which tell whether a character is upper or lowercase. Non-letters are considered neither uppercase nor lowercase.

```

>>> 'A'.isupper()
True
>>> 'a'.isupper()
False
>>> ' '.isupper()
False

```

To accommodate lowercase letters in the `rotate_char` function, we can add an `if/else` structure in order to rotate it one way (doing arithmetic relative to uppercase A) when uppercase, and to do something different when it is lowercase. Here is the new source file.

```

def rotate_char(ch): #Does capital or lowercase letters, but no non-letter symbols
    if ch.isupper():
        index = ord(ch)-ord('A')
        index = (index+13)%26 # **ROTATE** the letter 13 places
        return chr(ord('A')+index)
    else :
        index = ord(ch)-ord('a')
        index = (index+13)%26
        return chr(ord('a')+index)

def rotate_phrase(phrase):
    res = '' # Initially, the empty string
    for ch in phrase: #Rotate the characters, one by one
        res += rotate_char(ch)
    return res

```

Note that we do not have to modify `rotate_phrase`, because we're only altering the work that it outsources to `rotate_char`. Now `rotate_phrase` works for uppercase and lowercase letters.

```
>>> execfile('rot13.py')
>>> rotate_phrase('TheraininSpain')
'GurenvavaFcnva'
```

17 elif

Finally, for our code to accommodate non-letters, we should add one more block to the `rotate_char` function, which processes a non-letter properly (namely, by returning it unchanged). To do this requires one more keyword, `elif` (short for “else if”). The `elif` keyword allows you to consider three or more separate cases. Here’s how it works in our example.

```
def rotate_char(ch): #Works for all symbols, letters or otherwise
    if ch.isupper():
        index = ord(ch)-ord('A')
        index = (index+13)%26 # **ROTATE** the letter 13 places
        return chr(ord('A')+index)
    elif ch.islower() :
        index = ord(ch)-ord('a')
        index = (index+13)%26
        return chr(ord('a')+index)
    else:
        return ch #Don't change non-letters

def rotate_phrase(phrase):
    res = '' # Initially, the empty string
    for ch in phrase: #Rotate the characters, one by one
        res += rotate_char(ch)
    return res
```

You can include multiple `elif`s in a row if you wish. The interpreter will only execute one of the indented blocks of code – either the first one where the provided condition evaluated to “True,” or the “else” block if non of them do. Note that you can omit the “else” block if you wish, which has the same effect as leaving it empty.

Here’s a quick check that this worked.

```
>>> execfile('rot13.py')
>>> rotate_phrase('The rain in Spain')
'Gur enva va Fcnva'
```

18 Problem 4 solution

Everything looks functional. Now we can add the input/output code to the end of `rot13.py` and submit it to hackerrank. It will now pass all test cases.

```
def rotate_char(ch): #Works for all symbols, letters or otherwise
    if ch.isupper():
        index = ord(ch)-ord('A')
        index = (index+13)%26 # **ROTATE** the letter 13 places
        return chr(ord('A')+index)
    elif ch.islower() :
```

```
        index = ord(ch)-ord('a')
        index = (index+13)%26
        return chr(ord('a')+index)
    else:
        return ch #Don't change non-letters

def rotate_phrase(phrase):
    res = '' # Initially, the empty string
    for ch in phrase: #Rotate the characters, one by one
        res += rotate_char(ch)
    return res

line = raw_input()
print rotate_phrase(line)
```

```
>>> execfile('rot13.py')
The rain in Spain falls mainly on the plain.
Gur enva va Fcnva snyyf znvayl ba gur cynva.
```