Lab 2: Basic Loops and Functions

CSE/IT 107

NMT Department of Computer Science and Engineering

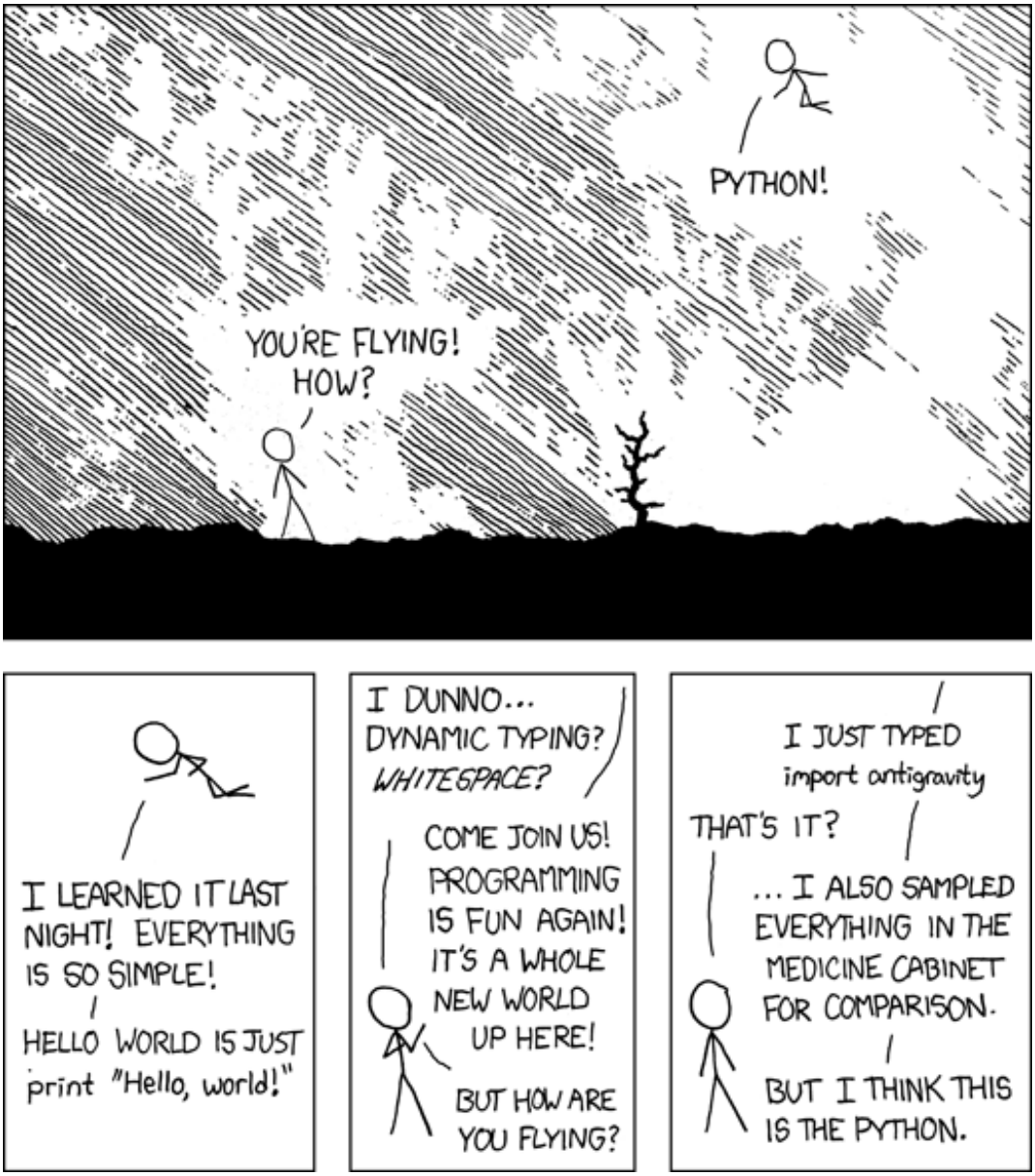


Figure 1: xkcd 353: Python (Source: `http://xkcd.com/353`)

"Only ugly languages become popular. Python is the exception."

— Donald Knuth

"Simplicity is the ultimate sophistication."

— Leonardo Da Vinci

"How do we convince people that in programming simplicity and clarity – in short: what mathematicians call elegance – are not a dispensable luxury, but a crucial matter that decides between success and failure?"

— Edsger Dijkstra

---

1 Introduction

The purpose of this lab is to expand upon the fundamentals of flow control. In the previous lab, we showed you how to do basic calculations in Python, like converting temperature from Celsius to Fahrenheit and Kelvin, as well as using `if`, `elif`, and `else` to execute different code depending on what values are given to the program.

In some of the exercises (like `star.py`), you likely needed to repeat lines of code in order to draw your shape correctly. Hopefully this struck you as something that would be nice to avoid: it might be sustainable with fairly simple shapes, but what if you wanted a 100-pointed star? Or what if you wanted to have the user type in a number to determine how many sides they wanted their shape to have? Clearly, repeating lines of code is not sustainable. In this lab, you will learn about the `while` statement to help with code you want to run several times.

We will also be learning about functions, which allow you to group your code into conveniently reusable chunks.

Contents

2 On Types

In Python, every value has a type. We have already seen a few types in action: integers, floating-point numbers, strings, and boolean values. The type of a value or variable restricts what it can represent. Here is a list of some types and example values.

Name	Description	Examples	Try to cast y to this type
`int`	Integer, whole number	`1, 123, -12,...`	`int(y)`
`float`	Floating-point number	`1.0, 3.1415, -0.01,...`	`float(y)`
`string`	Sequence of characters	`"", "1", "abc 123\n",...`	`str(y)`
`boolean`	True or false	`True, False`	`bool(y)`

Most programming languages have types for a good reason: for one, operations (such as `+`, `-`, ...) have different effects on different types. For example, an integer `*` an integer results in an integer (the multiplication of the two *operands*), but an integer `*` a string results in the string repeated. However, a string `*` a string results in an error:

```
>>> 5*3
15
>>> 5*'hi'
'hihihihihi'
>>> 'hi'*'hi'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't multiply sequence by non-int of type 'str'
```

In addition to that, some operations do not work on some types, which helps ensure correctness of your program.

Also, Python can speed up your programs by picking efficient ways of dealing with different types. Multiplying integers and multiplying floating point numbers require very different techniques. Knowing what kind of number is being used, Python can optimize that code.

2.1 The `type()` Function

Use the function `type()` to find out the type of a value or the type of the value of a specific variable.

```
>>> type(5)
<class 'int'>
>>> x = 10
>>> type(x)
<class 'int'>
>>> x = 'Allons-y!'
>>> type(x)
<class 'str'>
>>> type(True)
<class 'bool'>
>>> x = "5.5" # string containing 5.5
>>> y = float("5.5") # convert string with 5.5 to a float
```

```
>>> y
5.5
>>> z = int("5") # integer containing 5
>>> z
5
>>> q = str(5) # string containing 5
>>> q
'5'
>>> # Note that an impossible conversion will throw an error
>>> int("55a")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '55a'
```

2.2 Casting

We call the process of converting a value from one type to another type *casting*. Note that only values have types, and variables can hold different values. Applying `type()` to a variable will give you the type of the value that it holds at the time. In the previous section, we learned about comparison operators such as `!=`, `<`, `>`, etc. What was not mentioned is that these comparison operators also compare the type in some cases, but not in others:

```
>>> 5.5 == '5.5' # comparing a string to a float does not work
False
>>> x = 5.5
>>> y = 5.5
>>> x == y
True
>>> 5.5 == float('5.5')
True
>>> 5.0 == 5 # comparing a float and an int works
True
```

The usual rule for what works and what does not work with comparisons is to listen to your intuition: it makes sense to compare different types of numbers; while it does not make sense to compare strings with numbers.

2.3 Summary

- A type is a set of values that it can represent.
- Casting is the process of converting a value from one type to another.
- To find out what type a value / variable is, use the `type()` function.
- Types we know about at this point: `str`, `int`, `float`, `bool`
- Not coincidentally, these type names are also the names of the casting functions.

3 Formatting Strings with `.format()`

Previously, when we wanted to print out both a number and a string, we had to resort to this:

```
>>> x = 5
>>> print("x is equal to " + str(x))
x is equal to 5
```

However, there is an easier way to accomplish the same thing. By using the `.format()` command, as shown below, we can have far more options for how we format our output. With this we do not need to add a gap in the string and then use `+`.

```
>>> x = 5
>>> print("x is equal to {}".format(x))
x is equal to 5
```

This replaces `{}` with the value of `x`. If we include multiple instances of `{}` in our string, we can then pass multiple variables to `.format()`. It will place them in the string in the order provided.

```
>>> x = 5
>>> y = 6
>>> print("x is equal to {} and y is equal to {}.".format(x, y))
x is equal to 5 and y is equal to 6.
```

We can also use `.format()` to control our output. For example, we can restrict how many decimal places a floating point number is printed with. To do this, we add `:.2f` inside of the `{}`. The `.2f` specifies that we want 2 digits to follow the decimal point.

```
>>> print(3.141592653589793)
3.141592653589793
>>> print("{:.2f}".format(3.141592653589793))
3.14
```

For more format options, see `https://pyformat.info/` and
`https://docs.python.org/3/library/string.html#format-string-syntax`

3.1 Summary

- Syntax:

  ```
  print("string containing {}".format(variable))
  ```

  This will replace the `{}` with the value of `variable`.
- You can include multiple `{}` in a string and pass multiple values to `.format()`.
- You can specify advanced formatting options, such as number of digits after the decimal point.

4 Loops

4.1 While Loops

The syntax of a `while` loop is very similar to that of an `if` statement, but instead of only running the indented block of code once, the `while` loop will continue running it until the given boolean statement is no longer true.

```
x = 10
while x > 0:
    print(x)
    x = x - 1
```

The above program will print out the numbers 10 to 1. Try stepping through this program on paper, writing out the value of `x` at each time through the loop. Then repeat for this modified version of the program:

```
x = 10
while x > 0:
    x = x - 1
    print(x)
```

This version of the program will print out the numbers 9 to 0. This might seem a bit strange, since the condition of the loop says it will stop when `x` is no longer larger than 0. And yet, it prints out the value 0 before the loop ends. This is because the loop condition is only checked whenever the end of the indented section is reached. If the condition is `True`, then the indented section will be executed again. If the condition is `False`, then the loop will end.

If the condition starts out `False`, then the loop will never execute. The following program will not print anything:

```
x = 0
while x > 0:
    x = x - 1
    print(x)
```

`if` and `else` can be nested within `while`, as shown below:

```
x = 10
while x > 0:
    if x % 2 == 0:
        print("{} is even.".format(x))
    else:
        print("{} is odd.".format(x))
    x = x - 1
```

They can be nested the other way around, too, with a `while` inside conditional statements. Here's an easy way to start an infinite loop:

```
while True:
    print("Printing forever")
```

Press *Ctrl+C* to stop this loop.

4.2 Nesting

You can nest loops and conditional statements in any way you like. For example:

```
inputs = 0
while inputs < 10:
    parity = input("Even or odd? ")

    # prints even or odd numbers between 0 and 10, depending on user input
    if parity == "odd":
        n = 1
        while n < 11:
            if n % 2 == 1:
                print(n)
            n = n + 2
    elif parity == "even":
        n = 0
        while n <= 10:
            print(n)
            n = n + 2
    else:
        print("You did not enter even or odd.")
    inputs += 1
```

4.3 Summary

- Syntax:

  ```
  while condition:
      # code to be repeated
  ```

  This will repeat the indented code following the `while` until the condition is not true anymore. It checks the condition first, then runs the indented code, then checks the condition again, etc. Thus, if the condition is wrong in the first place, it will never run.

- There can be infinite while loops.

- You can nest conditional statements and loops any way you want in any combination.

5 `def`: Functions

So far, we have used functions such as `print()` and `math.sqrt()`, but we have not yet written our own functions.

Before we dive into that, let's talk about why to write functions. Some reasons:

- Instead of writing the same code again, we can just call a function containing the code again. (Functions are *reusable*.)
- Functions allow us to break our programs into many smaller pieces. This also allows us to easily think about each small piece in detail.
- Functions allow us to test small parts of our programs while not affecting other parts of the program – this reduces errors in our code.

A Python function is simply a "container" for a sequence of Python statements that do some task. Usually, functions are specialized to perform one clearly defined task. Here's the general form of how to write a function:

```
def function_name(arg0, arg1, ...):
    # block of code
```

A function can have *zero or more* arguments. For example:

```
>>> def pirate_noises():
...     i = 1
...     while i <= 4:
...         print("Arr!")
...         i += 1
...
```

5.1 Calling Functions

To call this function:

```
>>> pirate_noises()
Arr!
Arr!
Arr!
Arr!
```

To call a function, use its name followed by parentheses which contain comma-separated parameters:

```
function_name(param0, param1, ...)
```

- You must use parentheses both in the function definition and the function call, even if there are zero arguments. As a counterexample, typing `pirate_noises` without parentheses does not call that function.
- The parameter values are substituted for the corresponding arguments to the function. I.e.

  `param0` is replaced by value of `arg0`
  `param1` is replaced by value of `arg1`

  and so on. For example:

```
>>> def grocer(num_fruits, fruit_kind):
...     print('Stock: {} cases of {}'.format(num_fruits, fruit_kind))
...
>>> grocer(37, 'kale')
Stock: 37 cases of kale
>>> grocer(0, 'bananas')
Stock: 0 cases of bananas
```

5.2 `return`: Giving back values from a function

When we used functions from the math module, we were always able to assign the result of a function to a variable or to print it. For example:

```
>>> import math
>>> x = math.sqrt(16)
>>> print(x)
4.0
```

So how do we get a function to give back a value (*return* a value)? We use the return statement:

```
>>> def square(x):
...     return x**2
...
>>> y = square(5)
>>> print(y)
25
>>> square(4.3)
18.49
```

As soon as a `return` statement is reached, the function stops executing and just returns the value given to it. Any subsequent statements that are part of the function will be omitted. For example:

```
>>> def wage(hours, base_rate):
...     if hours > 40:
...         ot_pay = (hours - 40) * base_rate * 1.5
...         return base_rate * 40 + ot_pay
...     pay = hours * base_rate
```

```
...     return pay
...
>>> wage(40, 10)
400
>>> wage(50, 10)
550
```

- You can omit the expression after the return and just use a statement of this form to return the special value `None` is returned from the function:

```
return
```

- If Python executes your function body and never encounters a `return` statement, the effect is the same as a `return` with no value at the end of the function body: the special value `None` is returned. The `grocer` function returns `None`.

A function may also call other functions. If we keep using the wage example and add the ability to calculate the pay after taxes:

```
def wage(hours, base_rate):
    """Calculate and return weekly pay for a given amount of hours and base rate taking
    into consideration overtime pay at 1.5 times the given rate."""
    if hours > 40:
        ot_pay = (hours - 40) * base_rate * 1.5
        return base_rate * 40 + ot_pay
    pay = hours * base_rate
    return pay

def wage_after_tax(hours, base_rate, tax_rate):
    """Calculate and return weekly pay after taxes for a given amount of hours and a
    base rate with a flat tax rate."""
    pay = wage(hours, base_rate)
    return pay * (1 - tax_rate)
```

5.3 Summary

- Function definition syntax and function calling syntax:

```
def function_name(arg0, arg1, ...):
    # block of code
```

```
function_name(param0, param1, ...)
```

- A function may take zero or more arguments.
- A function returns one value. (If the programmer does not specify a value, the special value `None` is returned.)
- A good resource: `https://docs.python.org/3/tutorial/controlflow.html#defining-functions`

6 Making Calculations Shorter

We showed you simple Python operators such as `+`, `-`, `*`, `%`, etc in lab 1. There is a small extension to these that you can use to update a variable:

```
>>> x = 5
>>> x += 3 # same as x = x + 3
>>> x
8
```

The available assignment operators are:

`+=` addition	`*=` multiplication	`//=` integer division
`-=` subtraction	`/=` division	`**=` exponentiation

7 Exercises

How to Run Tests on Your Code

Some exercises have tests that you can run to make sure everything works properly. To run the test file `test_X.py`, which generally tests the file `X.py`, type

```
$ python3 test_X.py
```

Ensure `test_X.py` and `X.py` are in the same directory. If your code fails any of the tests, your output should look like:

```
$ python3 test_factorial.py
======================================================================
FAIL: test_factorial (__main__.FactorialTest) (i=20)
----------------------------------------------------------------------
Traceback (most recent call last):
  File ''test_factorial.py'', line 27, in test_factorial
    self.assertEqual(test_out, out_,
        msg=''input={}, expected output {}, received {}''\
        .format(in_, out_, test_out))
AssertionError: 0 != 2432902008176640000 :
    input=20, expected output 2432902008176640000, received 0

----------------------------------------------------------------------
Ran 1 test in 0.010s

FAILED (failures=11)
```

If your code passes all the tests, the output should look like:

```
$ python3 test_factorial.py
.
----------------------------------------------------------------------
Ran 1 test in 0.829s

OK
```

Exercise 7.1 (rps.py).
Write a program that reads a character for playing the game of rock-paper-scissors. If the character entered by the user is not one of "R", "P", or "S", the program keeps on prompting the user to enter a character. Once they enter a valid character, print out what they chose and quit.

For example:

```
Enter R, P, or S >>> A
Did not enter R, P, or S. Try again.
Enter R, P, or S >>> R
You chose rock. Exiting.
```

Exercise 7.2 (sums.py).
Write a program that keeps prompting the user for numbers to add to a sum until the user types in "exit". Then, display the sum of the numbers previously entered. Assume the user input is nothing other than numbers or "exit".

For example:

```
Enter a number to add to the sum >>> 15
Enter a number to add to the sum >>> 14.5
Enter a number to add to the sum >>> 12.25
Enter a number to add to the sum >>> exit
Sum of numbers: 41.75
```

Supplementary Files

A testing script is available on Canvas as `test_sums.py`.

Exercise 7.3 (fizzbuzz.py).
Have the user enter a positive integer number. Then, print the numbers from 1 to that number each on a line. When the printed number is divisible by 3, print "Fizz", and when the number is divisible by 5, print "Buzz", and when it is divisible by both, print "FizzBuzz".

You must use `.format()` and a `while` loop.

Should look like this when run:

```
Enter a number: -1
Not a positive number!
```

```
Enter a number: 16
1
2
3 Fizz
4
5 Buzz
6 Fizz
7
8
9 Fizz
10 Buzz
11
12 Fizz
13
14
15 FizzBuzz
16
```

Exercise 7.4 (calls.py).
You buy an international calling card to Germany. The calling card company has some special offers.

(a) If you charge your card with less than $10, you don't get anything extra.

(b) For a less than $25 charge, you get $3 of extra phone time.

(c) For a less than $50 charge, you get $8 of extra phone time.

(d) For a less than $100 charge, you get $20 of extra phone time.

(e) For a charge greater than or equal to $100, you get $25 of extra phone time.

Write a function that takes the value the user wants to charge and returns the actual value charged.

In your script, include a way for someone running the script to enter values to charge and get the actual value charged.

Example:

```
Enter value you want to charge >>> 24
27 dollars were added to your calling card.
```

Exercise 7.5 (primes.py).
Write a function that checks if a number N is prime. Your function should take in a single argument (the number to test) and return `True` or `False` depending on whether the number is prime or not.

Have your program ask the user for the number, then call the function you wrote to check if it is prime. If the number is prime, print `"prime"` otherwise print `"not prime"`.

Remember that a prime number is a number that is divisible only by 1 and itself. A simple approach checks all numbers from 2 up to N.

Try to improve on the simple approach, though: do we really need to check all those numbers? At which point do you know that you can stop? Remember to `import math`.

Note all primes are positive. There are an infinite number of primes. Some of the bigger primes are

$2^{74,207,281} - 1$	2,147,483,647	6,700,417
131,071	8,191	499

Sample output:

```
$ python3 primes.py
Enter a number: 2
prime
$ python3 primes.py
Enter a number: 4
not prime
$ python3 primes.py
Enter a number: -2
not prime
$ python3 primes.py
Enter a number: 499
prime
```

Supplementary Files

A testing script is available on Canvas as `test_primes.py`.

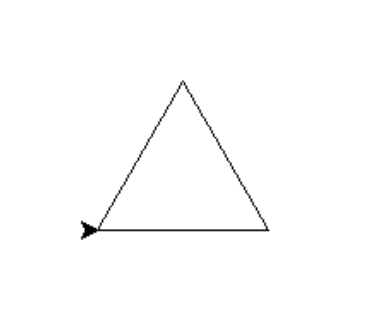
Exercise 7.6 (polygons2.py).
Write a program that takes in a number using `input` and then draws a regular polygon with that many sides. A regular polygon is one where each side is the same length and each corner is the same angle.

The code to draw the polygon should be entirely within a function that takes in a single integer as an argument.

Input:

```
How many sides? 3
```

Output:



8 Extra Credit Exercise

(This exercise is not required.)

Irrational numbers, like π, $\log 2$ or e do not terminate and are not generally easy to compute (unlike $1/9 = 0.1111\ldots$, for example.) In this exercise, you will write a Python script that can calculate the n^{th} digit of π without having to compute the first $n-1$ digits of π.

- Your program will find the n^{th} digit of π in hexadecimal base instead of the usual decimal base. For example, here are the first four digits after the decimal of π: $\pi = 3.14\ldots_{10} = 3 \cdot 10^0 + 1 \cdot 10^{-1} + 4 \cdot 10^{-2}$. In hexadecimal notation, instead of powers of 10, there are powers of 16 and digits go from 0 to A (which represents 10) and on to F (which represents 15) instead of 0 to 9. Here are some examples:

$$1A.0C_{16} = 1 \cdot 16^2 + 10 \cdot 16^1 + 0 \cdot 16^{-1} + 12 \cdot 16^{-2} \qquad 1A = 1 \cdot 16^2 + 10 \cdot 16^1$$

$$\pi = 3.243F\ldots_{16} = 3 \cdot 16^0 + 2 \cdot 16^{-1} + 4 \cdot 16^{-2} + 3 \cdot 16^{-3} + 15 \cdot 16^{-4} + \cdots$$

- Note the first 30 digits of π written in hexadecimal are equal to the first 36 digits of π in decimal:

$$\pi = 3.14159265358979323846264338327950288_{10}\ldots$$
$$\pi = 3.243F6A8885A308D313198A2E037073_{16}\ldots$$

- You will need to find the fractional part of a floating point number. This process is denoted using frac ( ). This function is not available in Python (hint: think about converting the float to an integer.)

$$\text{frac}\,(1.1) = 0.1 \qquad \text{frac}\,(123.1) = 0.1 \qquad \text{frac}\,(3.14) = 0.14$$

- Sigma notation is used to represent summations. Here are some examples:

$$\sum_{i=10}^{15} 2i = 20 + 22 + 24 + 26 + 28 + 30 \qquad \sum_{k=1}^{4} \frac{16^k}{8k+j} = \frac{16}{j} + \frac{16^2}{16+j} + \frac{16^3}{24+j} + \frac{16^4}{32+j}$$

Supplementary Files

A testing script is available on Canvas as `test_extra_nth_pi.py`.

The template `extra_nth_pi.py` is also provided.

Extra Credit Exercise 8.1 (extra_credit_pi.py).
Write a function `digits(n)` that finds the digits after the n^{th} digit of π in hexadecimal. A variation of the Bailey-Borwein-Plouffe can be used. First, we need a helper function S:

$$S(j,n) = \left(\sum_{k=0}^{n} \text{frac}\left(\frac{\left(16^{n-k}\right) \bmod (8k+j)}{8k+j}\right)\right) + \sum_{k=n+1}^{100} \text{frac}\left(\frac{16^{n-k}}{8k+j}\right)$$

The formula is:

$$\text{digits after } n^{\text{th}} \text{ digit of } \pi = \text{frac}\left(4S(1,n) - 2S(2,n) - S(5,n) - S(6,n)\right) + 1$$

The function `digits(n)` must return a string representing the hexadecimal digits at the n^{th} place of π. This string can be at most 10 characters in length (hint, use the function included in the template file: `decimal_to_hex(s, 10)`.) The code used to test this function relies on this name and behavior. It is highly recommended to use the included function `expm(b, p, a)` for efficiently calculating

$$(b^p) \bmod a = \texttt{(b ** p) \% a}.$$

```
$ python3 extra_nth_pi.py
Enter a number: 0
pi after digit 0 is 243F6A8885
$ python3 extra_nth_pi.py
Enter a number: 3
pi after digit 3 is F6A8885A30
$ python3 extra_nth_pi.py
Enter a number: 100
pi after digit 100 is 29B7C97C50
$ python3 extra_nth_pi.py
Enter a number: 1000000
pi after digit 1000000 is 6C65E52CB4
```

9 Submitting

You should submit your code as a tarball. It should contain all files used in the exercises for this lab. The submitted file should be named

`cse107_firstname_lastname_lab2.tar.gz`

Upload your tarball to Canvas.

List of Files to Submit

List of Files to Submit for Extra Credit