P4M_Notebook1

December 13, 2018

1 Part 1. Computing with Python 3

Welcome to programming!

What is the difference between *Python* and a calculator? We begin this first lesson by showing how Python can be used **as** a calculator, and we move into one of the most important programming structures -- the **loop**. Loops allow computers to carry out repetetive computations, with just a few commands.

1.1 Table of Contents

1.2 Python as a calculator

Different kinds of data are stored as different *types* in Python. For example, if you wish to work with integers, your data is typically stored as an *int*. A real number might be stored as a *float*. There are types for booleans (True/False data), strings (like "Hello World!"), and many more we will see.

A more complete reference for Python's numerical types and arithmetic operations can be found in the official Python documentation. The official Python tutorial is also a great place to start.

Python allows you to perform arithmetic operations: addition, subtraction, multiplication, and division, on numerical types. The operation symbols are `+`, `-`, `*`, and `/`. Evaluate each of the following cells to see how Python performs operations on *integers*. To evaluate the cell, click anywhere within the cell to select it (a selected cell will probably have a thick green line on its left side) and use the keyboard shortcut *Shift-Enter* to evaluate. As you go through this and later lessons, try to *predict* what will happen when you evaluate the cell before you hit Shift-Enter.

```
In [3]: 2 + 3
```

```
Out[3]: 5
```

```
In [4]: 2 * 3
```

```
Out[4]: 6

In [5]: 5 - 11

Out[5]: -6

In [6]: 5.0 - 11

Out[6]: -6.0

In [7]: 5 / 11

Out[7]: 0.45454545454545453

In [8]: 6 / 3

Out[8]: 2.0

In [9]: 5 // 11

Out[9]: 0

In [10]: 6 // 3

Out[10]: 2
```

The results are probably not too surprising, though the last two require a bit of explanation. Python *interprets* the input number 5 as an *int* (integer) and 5.0 as a *float*. "Float" stands for "floating point number," which are decimal approximations to real numbers. The word "float" refers to the fact that the decimal (or binary, for computers) point can float around (as in 1.2345 or 12.345 or 123.45 or 1234.5 or 0.00012345). There are deep computational issues related to how computers handle decimal approximations, and you can read about the IEEE standards if you're interested.

Python enables different kinds of division. The single-slash division in Python 3.x gives a floating point approximation of the quotient. That's why `5 / 11` and `6 / 3` both output floats. On the other hand, `5 // 11` and `6 // 3` yield integer outputs (rounding down) -- this is useful, but one has to be careful!

In fact the designers of Python changed their mind. **This tutorial assumes that you are using Python 3.x.** If you are using Python 2.x, the command `5 / 11` would output zero.

```
In [11]: -12 // 5  # What will this output?  Guess before evaluating!

Out[11]: -3
```

Why use integer division // and why use floating point division? In practice, integer division is typically a faster operation. So if you only need the rounded result (and that will often be the case), use integer division. It will run much faster than carrying out floating point division then manually rounding down.

Observe that floating point operations involve approximation. The result of `5.0/11.0` might not be what you expect in the last digit. Over time, especially with repeated operations, *floating point approximation* errors can add up!

You might be wondering about the little In[XX] and Out[XX] prompts. What is their purpose? Guess what the following line will do.

```
In [14]: Out[3] + Out[5]

Out[14]: -1
```

Cool, huh? It's nice to have a record of previous computations, especially if you don't want to type something again.

Python allows you to group expressions with parentheses, and follows the order of operations that you learn in school.

```
In [39]: (3 + 4) * 5

Out[39]: 35

In [40]: 3 + (4 * 5)

Out[40]: 23

In [41]: 3 + 4 * 5   #  What do you think will be the result?  Remember PEMDAS?

Out[41]: 23
```

Now is a good time to try a few computations of your own, in the empty cell below. You can type any Python commands you want in the empty cell. If you want to insert a new cell into this notebook, it takes two steps: 1. Click **to the left** of any existing cell. This should make a blue bar appear to the left of the cell. 2. Use the keyboard shortcut **a** to insert a new cell **above** the blue-selected cell or **b** to insert a new cell **below** the blue-selected cell. You can also use the keyboard shortcut **x** do delete a blue-selected cell... be careful!

```
In [18]: (3 + 8) * 12 - 8 % 6 #  An empty cell.  Have fun!

Out[18]: 130
```

For number theory, *division with remainder* is an operation of central importance. Integer division provides the quotient, and the operation % provides the remainder. It's a bit strange that the percent symbol is used for the remainder, but this dates at least to the early 1970s and has become standard across computer languages.

```
In [11]: 23 // 5  # Integer division

Out[11]: 4

In [10]: 23 % 5  # The remainder after division

Out[10]: 3
```

Note in the code above, there are little "comments". To place a short comment on a line of code, just put a hashtag # at the end of the line of code, followed by your comment.

Python gives a single command for division with remainder. Its output is a *tuple*.

```
In [12]: divmod(23,5)
```

```
Out[12]: (4, 3)

In [13]: type(divmod(23,5))

Out[13]: tuple
```

All data in Python has a type, but a common complaint about Python is that types are a bit concealed "under the hood". But they are not far under the hood! Anyone can find out the type of some data with a single command.

```
In [14]: type(3)

Out[14]: int

In [15]: type(3.0)

Out[15]: float

In [16]: type('Hello')

Out[16]: str

In [17]: type([1,2,3])

Out[17]: list
```

The key to careful computation in Python is always being *aware of the type* of your data, and *knowing* how Python operates differently on data of different types.

```
In [19]: 3 + 3

Out[19]: 6

In [20]: 3.0 + 3.0

Out[20]: 6.0

In [21]: 'Hello' + 'World!'

Out[21]: 'HelloWorld!'

In [22]: [1,2,3] + [4,5,6]

Out[22]: [1, 2, 3, 4, 5, 6]

In [23]: 3 + 3.0

Out[23]: 6.0

In [24]: 3 + 'Hello!'  # Uh oh!
```

```
        ---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call last)

        <ipython-input-24-dacf80a979a3> in <module>()
    ----> 1 3 + 'Hello!'  # Uh oh!


        TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

```
In [28]: #  An empty cell.  Have fun!
         #  Try operating on ints, floats, and strings, with different operations.  Which ones
         (4,3) + (5,6)

Out[28]: (4, 3, 5, 6)

In [32]: 6.66 + 3.34

Out[32]: 10.0
```

As you can see, addition (the + operator) is interpreted differently in the contexts of numbers, strings, and lists. The designers of Python allowed us to add *numbers* of different types: if you try to operate on an *int* and a *float*, the *int* will typically be *coerced* into a float in order to perform the operation. But the designers of Python did not give meaning to the addition of a number with a string, for example. That's why you probably received a *TypeError* after trying to add a number to a string.

On the other hand, Python does interpret *multiplication* of a natural number with a string or a list.

```
In [35]: 3 * 'Hello!'

Out[35]: 'Hello!Hello!Hello!'

In [36]: 0 * 'Hello!'

Out[36]: ''

In [37]: 2 * [1,2,3]

Out[37]: [1, 2, 3, 1, 2, 3]
```

Can you create a string with 100 A's (like `AAA...`)? Use an appropriate operation in the cell below.

```
In [38]:  100 * 'A'#  Practice cell

Out[38]: 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

Exponents in Python are given by the ** operator. The following lines compute 2 to the 1000th power, in two different ways.

```
In [33]: 2**1000

Out[33]: 10715086071862673209484250490600018105614048117055336074437503883703510511249361224931

In [34]: 2.0**1000

Out[34]: 1.0715086071862673e+301
```

As before, Python interprets an operation (**) differently in different contexts. When given integer input, Python evaluates `2**1000` **exactly**. The result is a large integer. A nice fact about Python, for mathematicians, is that it handles exact integers of arbitrary length! Many other programming languages (like C++) will give an error message if integers get too large in the midst of a computation.

New in version 3.x, Python implements long integers without giving signals to the programmer or changing types. In Python 2.x, there were two types: *int* for somewhat small integers (e.g., up to 2^{31}) and *long* type for all larger integers. Python 2.x would signal which type of integer was being used, by placing the letter "L" at the end of a long integer. Now, in Python 3.x, the programmer doesn't really see the difference. There is only the *int* type. But Python still optimizes computations, using hardware functionality for arithmetic of small integers and custom routines for large integers. The programmer doesn't have to worry about it most of the time.

For scientific applications, one often wants to keep track of only a certain number of significant digits (sig figs). If one computes the floating point exponent `2.0**1000`, the result is a decimal approximation. It is still a float. The expression "e+301" stands for "multiplied by 10 to the 301st power", i.e., Python uses *scientific notation* for large floats.

```
In [42]: type(2**1000)

Out[42]: int

In [43]: type(2.0**1000)

Out[43]: float

In [44]: type(divmod(2**26,12))#  An empty cell.  Have fun!

Out[44]: tuple
```

Now is a good time for reflection. Double-click in the cell below to answer the given questions. Cells like this one are used for text rather than Python code. Text is entered using *markdown*, but you can typically just enter text as you would in any text editor without problems. Press *shift-Enter* after editing a markdown cell to complete the editing process.

Note that a dropdown menu in the toolbar above the notebook allows you to choose whether a cell is Markdown or Code (or a few other things), if you want to add or remove markdown/code cells.

1.2.1 Exercises

1. What data types have you seen, and what kinds of data are they used for? Can you remember them without looking back?
2. How is division / interpreted differently for different types of data?
3. How is multiplication * interpreted differently for different types of data?
4. What is the difference between 100 and 100.0, for Python?

Double-click this markdown cell to edit it, and answer the exercises. This may be graded, so please complete all questions! Write in clear, complete, and concise sentences.

1. int, float, tuple, string, list. Ints are integers, floats are decimal numbers, a tuple is a sort of list, a string is a string of characters, a list is an array.
2. Division of integers results in an integer, division of an int and float results in a float, division of strings and characters results in an error.
3. Multiplication of ints results in an int, multiplication of floats results in a float, multiplication of an int and a float is a float, multiplication of a string and an int or float results in that many iterations of the string.
4. 100 is an int and 100.0 is a float.

1.3 Calculating with booleans

A *boolean* (type *bool*) is the smallest possible piece of data. While an *int* can be any integer, positive or negative, a *boolean* can only be one of two things: *True* or *False*. In this way, booleans are useful for storing the answers to yes/no questions.

Questions about (in)equality of numbers are answered in Python by *operations* with numerical input and boolean output. Here are some examples. A more complete reference is in the official Python documentation.

```
In [56]: 3 > 2
```

```
Out[56]: True
```

```
In [57]: type(3 > 2)
```

```
Out[57]: bool
```

```
In [58]: 10 < 3
```

```
Out[58]: False
```

```
In [59]: 2.4 < 2.4000001
```

```
Out[59]: True
```

```
In [60]: 32 >= 32
```

```
Out[60]: True

In [61]: 32 >= 31

Out[61]: True

In [62]: 2 + 2 == 4

Out[62]: True
```

Which number is bigger: 23^{32} or 32^{23}? Use the cell below to answer the question!

```
In [63]: (23**32) > (32**23)#  Write your code here.

Out[63]: True
```

The expressions <, >, <=, >= are interpreted here as **operations** with numerical input and boolean output. The symbol == (two equal symbols!) gives a True result if the numbers are equal, and False if the numbers are not equal. An extremely common typo is to confuse = with ==. But the single equality symbol = has an entirely different meaning, as we shall see.

Using the remainder operator % and equality, we obtain a divisibility test.

```
In [ ]: 63 % 7 == 0  # Is 63 divisible by 7?

In [ ]: 101 % 2 == 0  # Is 101 even?
```

Use the cell below to determine whether 1234567890 is divisible by 3.

```
In [64]: 1234567890 % 3 == 0 # Your code goes here.

Out[64]: True
```

Booleans can be operated on by the standard logical operations: and, or, not. In ordinary English usage, "and" and "or" are conjunctions, while here in *Boolean algebra*, "and" and "or" are operations with Boolean inputs and Boolean output. The precise meanings of "and" and "or" are given by the following **truth tables**.

and	True	False
True	True	False
False	False	False

or	True	False
True	True	True
False	True	False

```
In [65]: True and False
```

```
Out[65]: False

In [66]: True or False

Out[66]: True

In [67]: True or True

Out[67]: True

In [68]: not True

Out[68]: False
```

Use the truth tables to predict the result (True or False) of each of the following, before evaluating the code.

```
In [74]: (2 > 3) and (3 > 2)

Out[74]: False

In [75]: (1 + 1 == 2) or (1 + 1 == 3)

Out[75]: True

In [76]: not (-1 + 1 >= 0)

Out[76]: False

In [77]: 2 + 2 == 4

Out[77]: True

In [78]: 2 + 2 != 4  # For "not equal", Python uses the operation `!=`.

Out[78]: False

In [79]: 2 + 2 != 5  # Is 2+2 *not* equal to 5?

Out[79]: True

In [80]: not (2 + 2 == 5)  # The same as above, but a bit longer to write.

Out[80]: True
```

Experiment below to see how Python handles a double or triple negative, i.e., something with a `not not`.

```
In [81]: not not not True# Experiment here.

Out[81]: False
```

Python does give an interpretation to arithmetic operations with booleans and numbers. Try to guess this interpretation with the following examples. Change the examples to experiment!

```
In [73]: False - 100

Out[73]: -100

In [72]: True + 20

Out[72]: 21
```

This ability of Python to interpret operations based on context is a mixed blessing. On one hand, it leads to handy shortcuts -- quick ways of writing complicated programs. On the other hand, it can lead to code that is harder to read, especially for a Python novice. Good programmers aim for code that is easy to read, not just short!

The Zen of Python is a series of 20 aphorisms for Python programmers. The first seven are below.

> Beautiful is better than ugly.
>
> Explicit is better than implicit.
>
> Simple is better than complex.
>
> Complex is better than complicated.
>
> Flat is better than nested.
>
> Sparse is better than dense.
>
> Readability counts.

1.3.1 Exercises

1. Did you look at the truth tables closely? Can you remember, from memory, what `True or False` equals, or what `True and False` equals?

2. How might you easily remember the truth tables? How do they resemble the standard English usage of the words "and" and "or"?

3. If you wanted to know whether a number, like 2349872348723, is a multiple of 7 but **not** a multiple of 11, how might you write this in one line of Python code?

4. You can chain together `and` commands, e.g., with an expression like `True and True and True` (which would evaluate to `True`). You can also group booleans, e.g., with `True and (True or False)`. Experiment to figure out the order of operations (`and`, `or`, `not`) for booleans.

5. The operation `xor` means "exclusive or". Its truth table is: `True xor True = False` and `False xor False = False` and `True xor False = True` and `False xor True = True`. How might you implement `xor` in terms of the usual `and`, `or`, and `not`?

1.3.2 Solutions

(Edit here to give solutions to the exercises)

1. `True or False == True`, `True and False == False`
2. For `and` to be true, expression must be true. For `or` to be true, only one expression has to be true.
3. `2349872348723 % 7 == 0 and 2349872348723 % 11 != 0`
4. First `not` is applied to individual bools, then from left to right `and` and `or` are evaulated depending on what comes first.
5. Let `a` and `b` be boolean varialbes: `(a or b) and not(a and b)`

1.4 Declaring variables

A central feature of programming is the declaration of variables. When you declare a variable, you are *storing* data in the computer's *memory* and you are assigning a *name* to that data. Both storage and name-assignment are carried out with the *single* equality symbol =.

```
In [88]: e = 2.71828
```

With this command, the float 2.71828 is stored somewhere inside your computer, and Python can access this stored number by the name "e" thereafter. So if you want to compute "e squared", a single command will do.

```
In [89]: e * e

Out[89]: 7.3890461584

In [90]: type(e)

Out[90]: float
```

You can use just about any name you want for a variable, but your name *must* start with a letter, *must* not contain spaces, and your name *must* not be an existing Python word. Characters in a variable name can include letters (uppercase and lowercase) and numbers and underscores `_`.

So `e` is a valid name for a variable, but `type` is a bad name. It is very tempting for beginners to use very short abbreviation-style names for variables (like `dx` or `vbn`). But resist that temptation and use more descriptive names for variables, like `difference_x` or `very_big_number`. This will make your code readable by you and others!

There are different style conventions for variable names. We use lowercase names, with underscores separating words, roughly following Google's style conventions for Python code.

```
In [91]: my_number = 17

In [92]: my_number < 23

Out[92]: True
```

After you declare a variable, its value remains the same until it is changed. You can change the value of a variable with a simple assignment. After the above lines, the value of my_number is 17.

```
In [93]: my_number = 3.14
```

This command reassigns the value of my_number to 3.14. Note that it changes the type too! It effectively overrides the previous value and replaces it with the new value.

Often it is useful to change the value of a variable *incrementally* or *recursively*. Python, like many programming languages, allows one to assign variables in a self-referential way. What do you think the value of S will be after the following four lines?

```
In [94]: S = 0
         S = S + 1
         S = S + 2
         S = S + 3
         print(S)
```

```
6
```

The first line `S = 0` is the initial declaration: the value 0 is stored in memory, and the name S is assigned to this value.

The next line `S = S + 1` looks like nonsense, as an algebraic sentence. But reading = as **assignment** rather than **equality**, you should read the line `S = S + 1` as assigning the *value* `S + 1` to the *name* `S`. When Python interprets `S = S + 1`, it carries out the following steps.

1. Compute the value of the right side, `S+1`. (The value is 1, since `S` was assigned the value 0 in the previous line.)
2. Assign this value to the left side, `S`. (Now `S` has the value 1.)

Well, this is a slight lie. Python probably does something more efficient, when given the command `S = S + 1`, since such operations are hard-wired in the computer and the Python interpreter is smart enough to take the most efficient route. But at this level, it is most useful to think of a self-referential assignment of the form `X = expression(X)` as a two step process as above.

1. Compute the value of `expression(X)`.
2. Assign this value to `X`.

Now consider the following three commands.

```
In [95]: my_number = 17
         new_number = my_number + 1
         my_number = 3.14
```

What are the values of the variables my_number and new_number, after the execution of these three lines?

To access these values, you can use the *print* function.

```
In [96]: print(my_number)
         print(new_number)
```

```
3.14
18
```

Python is an *interpreted* language, which carries out commands line-by-line from top to bottom. So consider the three lines

```
my_number = 17
new_number = my_number + 1
my_number = 3.14
```

Line 1 sets the value of my_number to 17. Line 2 sets the value of new_number to 18. Line 3 sets the value of my_number to 3.14. But Line 3 does *not* change the value of new_number at all.

(This will become confusing and complicated later, as we study mutable and immutable types.)

1.4.1 Exercises

1. What is the difference between = and == in the Python language?
2. If the variable `x` has value `3`, and you then evaluate the Python command `x = x * x`, what will be the value of `x` after evaluation?
3. Imagine you have two variables `a` and `b`, and you want to switch their values. How could you do this in Python?

1.4.2 Solutions

(Use this space to work on the exercises.)

1. = assigns the left side to the right side, == compares the left side and right side and outputs a bool.
2. 9
3. `c = a a = b b = c`

1.5 Lists and ranges

Python stands out for the central role played by *lists*. A *list* is what it sounds like -- a list of data. Data within a list can be of any type. Multiple types are possible within the same list! The basic syntax for a list is to use brackets to enclose the list items and commas to separate the list items.

```
In [97]: type([1,2,3])

Out[97]: list

In [98]: type(['Hello',17])

Out[98]: list
```

There is another type called a *tuple* that we will use less often. Tuples use parentheses for enclosure instead of brackets.

```
In [99]: type((1,2,3))

Out[99]: tuple
```

There’s another list-like type in Python 3, called the `range` type. Ranges are kind of like lists, but instead of plunking every item into a slot of memory, ranges just have to remember three integers: their *start*, their *stop*, and their *step*.

The `range` command creates a range with a given start, stop, and step. If you only input one number, the range will ***start at zero*** and use ***steps of one*** and will stop ***just before*** the given stop-number.

One can create a list from a range (plunking every term in the range into a slot of memory), by using the `list` command. Here are a few examples.

```
In [100]: type(range(10)) # Ranges are their own type, in Python 3.x.  Not in Python 2.x!

Out[100]: range

In [101]: list(range(10)) # Let's see what's in the range.  Note it starts at zero!  Where doe

Out[101]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

A more complicated two-input form of the range command produces a range of integers **starting at** a given number, and **terminating before** another given number.

```
In [102]: list(range(3,10))

Out[102]: [3, 4, 5, 6, 7, 8, 9]

In [103]: list(range(-4,5))

Out[103]: [-4, -3, -2, -1, 0, 1, 2, 3, 4]
```

This is a common source of difficulty for Python beginners. While the first parameter (-4) is the starting point of the list, the list ends just before the second parameter (5). This takes some getting used to, but experienced Python programmers grow to like this convention.

The *length* of a list can be accessed by the len command.

```
In [104]: len([2,4,6])

Out[104]: 3

In [105]: len(range(10))  # The len command can deal with lists and ranges.  No need to conver

Out[105]: 10

In [106]: len(range(10,100)) # Can you figure out the length, before evaluating?

Out[106]: 90
```

The final variant of the range command (for now) is the *three-parameter* command of the form `range(a,b,s)`. This produces a list like `range(a,b)`, but with a "step size" of `s`. In other words, it produces a list of integers, beginning at `a`, increasing by `s` from one entry to the next, and going up to (but not including) `b`. It is best to experiment a bit to get the feel for it!

```
In [107]: list(range(1,10,2))

Out[107]: [1, 3, 5, 7, 9]

In [108]: list(range(11,30,2))

Out[108]: [11, 13, 15, 17, 19, 21, 23, 25, 27, 29]

In [109]: list(range(-4,5,3))

Out[109]: [-4, -1, 2]

In [110]: list(range(10,100,17))

Out[110]: [10, 27, 44, 61, 78, 95]
```

This can be used for descending ranges too, and observe that the final number b in range(a,b,s) is not included.

```
In [111]: list(range(10,0,-1))

Out[111]: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

How many multiples of 7 are between 10 and 100? We can find out pretty quickly with the range command and the len command (to count).

```
In [114]: len(list(range(10,100,7)))  # What list will this create?  It won't answer the quest

Out[114]: 13

In [116]: len(list(range(14,100,7)))  # Starting at 14 gives the multiples of 7.

Out[116]: 13

In [117]: len(range(14,100,7))  # Gives the length of the list, and answers the question!

Out[117]: 13
```

1.5.1 Exercises

1. If `a` and `b` are integers, what is the length of `range(a,b)`? Express your answer as a formula involving `a` and `b`.
2. Use a list and range command to produce the list `[1,2,3,4,5,6,7,8,9,10]`.
3. Create the list [1,2,3,4,5,1,2,3,4,5,1,2,3,4,5,1,2,3,4,5,1,2,3,4,5] with a single list and range command and another operation.
4. How many multiples of 3 are there between 300 and 3000?

```
In [121]: # 1. a-b-1
          #  Use this space to work on the exercises.
```

```
Out[121]: [1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5]
```

```
In [123]: #2.
          list(range(1,11))
```

```
Out[123]: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
In [124]: #3.
          list(range(1,6)) * 5
```

```
Out[124]: [1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5]
```

```
In [143]: #4.
          len(list(range(300,3000,3)))
```

```
Out[143]: 900
```

1.6 Iterating over a range

Computers are excellent at repetitive reliable tasks. If we wish to perform a similar computation, many times over, a computer a great tool. Here we look at a common and simple way to carry out a repetetive computation: the "for loop". The "for loop" *iterates* through items in a list or range, carrying out some action for each item. Two examples will illustrate.

```
In [126]: for n in [1,2,3,4,5]:
              print(n*n)
```

```
1
4
9
16
25
```

```
In [127]: for s in ['I','Am','Python']:
              print(s + "!")
```

```
I!
Am!
Python!
```

The first loop, **unraveled**, carries out the following sequence of commands.

```
In [128]: n = 1
          print(n*n)
          n = 2
          print(n*n)
          n = 3
          print(n*n)
          n = 4
          print(n*n)
          n = 5
          print(n*n)
```

```
1
4
9
16
25
```

But the "for loop" is more efficient *and* more readable to programmers. Indeed, it saves the repetition of writing the same command `print n*n` over and over again. It also makes transparent, from the beginning, the range of values that `n` is assigned to.

When you read and write "for loops", you should consider how they look unravelled -- that is how Python will carry out the loop. And when you find yourself faced with a repetetive task, you might consider whether it may be wrapped up in a for loop.

Try to unravel the loop below, and predict the result, before evaluating the code.

```
In [144]: P = 1
          for n in range(1,6):
              P = P * n
          print(P)
```

```
120
```

This might have been difficult! So what if you want to trace through the loop, as it goes? Sometimes, especially when debugging, it's useful to inspect every step of the loop to see what Python is doing. We can inspect the loop above, by inserting a print command within the *scope* of the loop.

```
In [146]: P = 1
          for n in range(1,6):
              P = P * n
              print("n is",n,"and P is",P)
          print(P)
```

```
n is 1 and P is 1
n is 2 and P is 2
n is 3 and P is 6
n is 4 and P is 24
n is 5 and P is 120
120
```

Here we have used the *print* command with strings and numbers together. In Python 3.x, you can print multiple things on the same line by separating them by commas. The "things" can be strings (enclosed by single or double-quotes) and numbers (int, float, etc.).

```
In [147]: print("My favorite number is",17)

My favorite number is 17
```

If we unravel the loop above, the linear sequence of commands interpreted by Python is the following.

```
In [148]: P = 1
          n = 1
          P = P * n
          print("n is",n,"and P is",P)
          n = 2
          P = P * n
          print("n is",n,"and P is",P)
          n = 3
          P = P * n
          print("n is",n,"and P is",P)
          n = 4
          P = P * n
          print("n is",n,"and P is",P)
          n = 5
          P = P * n
          print("n is",n,"and P is",P)
          print (P)

n is 1 and P is 1
n is 2 and P is 2
n is 3 and P is 6
n is 4 and P is 24
n is 5 and P is 120
120
```

Let's analyze the loop syntax in more detail.

```
P = 1
for n in range(1,6):
    P = P * n  # this command is in the scope of the loop.
    print("n is",n,"and P is",P)  # this command is in the scope of the loop too!
print(P)
```

The "for" command ends with a colon `:`, and the **next two** lines are indented. The colon and indentation are indicators of **scope**. The *scope* of the for loop begins after the colon, and includes all indented lines. The *scope* of the for loop is what is repeated in every step of the loop (in addition to the reassignment of `n`).

```
In [149]: P = 1
          for n in range(1,6):
```

```
        P = P * n  # this command is in the scope of the loop.
        print("n is",n,"and P is",P)  # this command is in the scope of the loop too!
    print(P)
```

```
n is 1 and P is 1
n is 2 and P is 2
n is 3 and P is 6
n is 4 and P is 24
n is 5 and P is 120
120
```

If we change the indentation, it changes the scope of the for loop. Predict what the following loop will do, by unraveling, before evaluating it.

```
In [150]: P = 1
          for n in range(1,6):
              P = P * n
          print("n is",n,"and P is",P)
          print(P)
```

```
n is 5 and P is 120
120
```

Scopes can be nested by nesting indentation. What do you think the following loop will do? Can you unravel it?

```
In [151]: for x in [1,2,3]:
              for y in ['a', 'b']:
                  print(x,y)
```

```
1 a
1 b
2 a
2 b
3 a
3 b
```

How might you create a nested loop which prints `1 a` then `2 a` then `3 a` then `1 b` then `2 b` then `3 b`? Try it below.

```
In [153]: # Insert your loop here.
          for x in ['a','b']:
              for y in [1,2,3,]:
                  print(y,x)
```

```
1 a
2 a
3 a
1 b
2 b
3 b
```

Among popular programming languages, Python is particular about indentation. Other languages indicate scope with open/close braces, for example, and indentation is just a matter of style. By requiring indentation to indicate scope, Python effectively removes the need for open/close braces, and enforces a readable style.

We have now encountered data types, operations, variables, and loops. Taken together, these are powerful tools for computation! Now complete the following exercises for more practice.

1.7 Exercises

1. Describe how Python interprets division with remainder when the divisor and/or dividend is negative.
2. What is the remainder when 2^{90} is divided by 91?
3. How many multiples of 13 are there between 1 and 1000?
4. How many *odd* multiples of 13 are there between 1 and 1000?
5. What is the sum of the numbers from 1 to 1000?
6. What is the sum of the squares, from $1 \cdot 1$ to $1000 \cdot 1000$?

```
In [ ]: # Insert your solutions here.
```

```
In [154]: #2.
          2**90 % 91
```

```
Out[154]: 64
```

```
In [155]: #3.
          len(list(range(1,1000,13)))
```

```
Out[155]: 77
```

```
In [162]: #4.
          total = 0
          for x in range(1,1000,13):
              if x % 2 != 0:
                  total = total + 1
          print(total)
```

```
39
```

```
In [158]: #5.
          sum(range(1,1000))
```

```
Out[158]: 499500
```

```
In [166]: #6.
          total = 0
          for x in range(1,1001):
              total = total + x**2
          print(total)
```

```
333833500
```

2 Explorations

Now that you have learned the basics of computation in Pytho and loops, we can start exploring some interesting mathematics! We are going to look at approximation here -- some ancient questions made easier with programming.

2.1 Exploration 1: Approximating square roots.

We have seen how Python can do basic arithmetic -- addition, subtraction, multiplication, and division. But what about other functions, like the square root? In fact, Python offers a few functions for the square root, but that's not the point. How can we compute the square root using only basic arithmetic?

Why might we care?

1. We might want to know the square root of a number with more precision than the Python function offers.
2. We might want to understand how the square root is computed... under the hood.
3. Understanding approximations of square roots and other functions is important, because we might want to approximate other functions in the future (that aren't pre-programmed for us).

Here is a method for approximating the square root of a number X.

1. Begin with a guess g.
2. Observe that g * (X / g) = X. Therefore, among the two numbers g and (X/g), one will be less than or equal to the square root of X, and the other will be greater than or equal to the square root.
3. Take the average of g and (X/g). This will be closer to the square root than g or X/g (unless your guess is exactly right!)
4. Use this average as a new guess... and go back to the beginning.

Now implement this in Python to approximate the square root of 2. Use a loop, so that you can go through the approximation process 10 times or 100 times or however many you wish. Explore the effect of different starting guesses. Would a change in the averaging function improve the approximation? How quickly does this converge? How does this change if you try square roots of different positive numbers?

Write your code (Python) and findings (in Markdown cells) in a readable form. Answer the questions in complete sentences.

```
In [112]: #  Start your explorations here!
          X = 2
          g = X/4

          for y in range(100):
              average = (g + (X/g))/2

              # break once a certain precision is reached
              if (abs(average - g) < 0.00000000000001):
                  break

              g = average
              print('Step', y, '...', g)
```

```
Step 0 ... 2.25
Step 1 ... 1.5694444444444444
Step 2 ... 1.4218903638151426
Step 3 ... 1.4142342859400734
Step 4 ... 1.4142135625249321
Step 5 ... 1.414213562373095
```

Answer:
A change in the averaging function would not improve the approximation. What would improve the algorithm is changing the value of the upper bound at each iteration, as I have it now, the upper bound remains the value of X the whole time as the lower bound converges.

3 Exploration 2: Approximating e and pi.

Now we approximate two of the most important constants in mathematics: e and pi. There are multiple approaches, but e is pretty easy with the series expansion of e^x. First, approximate e by the series expansion of e^x at x=1. How many terms are necessary before the float stabilizes? Use a loop, with a running product for the factorials and running sums for the series.

```
In [5]: #  Approximate e here.
        from math import factorial

        e = 1
        for y in range(100):
            old_e = e
            e = e + 1/factorial(y+1)

            # break once a certain precision is reached
            if ((e - old_e) < 0.00000000001):
                break

            print('Step', y, '...', e)
```

```
Step 0 ... 2.0
Step 1 ... 2.5
Step 2 ... 2.6666666666666665
Step 3 ... 2.708333333333333
Step 4 ... 2.7166666666666663
Step 5 ... 2.7180555555555554
Step 6 ... 2.7182539682539684
Step 7 ... 2.71827876984127
Step 8 ... 2.7182815255731922
Step 9 ... 2.7182818011463845
Step 10 ... 2.718281826198493
Step 11 ... 2.7182818282861687
Step 12 ... 2.7182818284467594
Step 13 ... 2.71828182845823
```

Answer:

Where the function stabilizes depends on how many digits of precision are wanted. In this case, it's accurate to 5 decimal places within 8 iterations.

Next we will approximate pi, which is much more interesting. We can try a few approaches. For a series-approach (like e), we need a series that converges to pi. A simple example is the arctangent atan(x). Recall (precalculus!) that atan(1) = pi/4. Moreover, the derivative of atan(x) is 1 / (1+x^2).

1. Figure out the Taylor series of 1 / (1+x^2) near x=0. Note that this is a geometric series!
2. Figure out the Taylor series of atan(x) near x=0 by taking the antiderivative, term by term, of the above.
3. Try to estimate pi with this series, using many terms of the series.

```
In [2]: #  Using ray tracing method to approximate pi
        from random import random

        in_circle = 0
        n = 1000000

        for r in range(n):
            for b in range(2):
                x = random()
            for b in range(2):
                y = random()

            if ((x**2 + y**2) <= 1):
                in_circle = in_circle + 1

        print(4 * (in_circle/n))
```

```
3.144948
```

```
In [3]: # testing Taylor series expansion of 1/(1+x^2)

        sum = 0
        x = 0.000001
        for n in range(100):
            sum = sum + (-1)**n * (x**(2*n))

        print('Taylor series ...', sum)
        print('f(x) ...', 1/(1+x**2))
```

```
Taylor series ... 0.999999999999
f(x) ... 0.9999999999989999
```

```
In [21]: # testing Taylor series expansion of arctan(x) at x=1 then multiply by 4 to get pi

         from math import pi
         sum = 0
         x = 1
         for n in range(10000):
             # this is the general forula of the Taylor series of arctan(x)
             sum = sum + (-1)**n * ((x**((2*n)+1)/(2*n+1)))

         print('estimated pi ...', sum*4)
         print('real pi (from math library) ...', pi)
         print('difference ... {:0.10f}'.format(abs((sum*4-pi))))
```

```
estimated pi ... 3.1414926535900345
real pi (from math library) ... 3.141592653589793
difference ... 0.0001000000
```

Now we'll accelerate things a bit. There's a famous formula of Machin (1706) who computed the first hundred digits of pi. We'll use his identity:

pi/4 = 4 * atan(1/5) - atan(1 / 239).

This isn't obvious, but there's a tedious proof using sum/difference identities in trig.

Try using this formula now to approximate pi, using your Taylor series for atan(x). It should require fewer terms.

```
In [60]: # using pi/4 = 4 * atan(1/5) - atan(1 / 239) formula
         # note this method is better than the previous, even with just a single iteration
         from math import pi
         from mpmath import *
         mp.prec = 100
         mp.dps = 100
```

```
        atan_15 = 0
        atan_1239 = 0
        x = 1/5
        y = 1/239
        for n in range(100000):
            atan_15 = atan_15 + (-1)**n * ((x**((2*n)+1)/(2*n+1)))
            atan_1239 = atan_1239 + (-1)**n * ((y**((2*n)+1)/(2*n+1)))

        pi_apprx = 4* (4 * atan_15 - atan_1239)
        print(atan_15)
        print(atan_1239)
        print('estimated pi ...', pi_apprx)
        print('real pi (from math library) ...', pi)
        print('difference ... ', abs((pi_apprx-pi)))
```

```
0.1973955598498808
0.004184076002074725
estimated pi ... 3.141592653589794
real pi (from math library) ... 3.14159265358979323846264338327950288419716939937510582097494
difference ...  0.0000000000000007657137397853899146162987412451561441790250554076921835937137
```

Now let's compare this to **Archimedes' method**. Archimedes approximated pi by looking at the perimeters p(n) and P(n) of (2ˆn)-gons inscribed in and circumscribed around a unit circle. So p(2) is the perimeter of a square inscribed in the unit circle. P(2) is the perimeter of a square circumscribed around a unit circle.

Archimedes proved the following (not in the formulaic language of algebra): For all n >= 2,
(P-formula) P(n+1) = 2 * p(n) * P(n) / (p(n) + P(n)).
(p-formula) p(n+1) = sqrt(p(n) * P(n+1)).

1. Compute p(2) and P(2).

2. Use these formulas to compute p(10) and P(10). Use this to get a good approximation for pi!

We could use our previous sqrt function if you want, we'll take a fancier high-precision approach. "mpmath" is a Python package for high-precision calculation. It should come with your Anaconda installation. You can read the full documentation at http://mpmath.org/doc/current/

First we load the package and print its status.

```
In [24]: from mpmath import *
         print(mp)
```

```
Mpmath settings:
  mp.prec = 53                [default: 53]
  mp.dps = 15                 [default: 15]
  mp.trap_complex = False     [default: False]
```

The number mp.dps is (roughly) the number of decimal digits that mpmath will keep track of in its computations. mp.prec is the binary precision, a bit more than 3 times the decimal precision. We can change this to whatever we want.

```
In [25]: mp.dps = 50 # Let's try 50 digits precision to start.
         print(mp)
```

```
Mpmath settings:
  mp.prec = 169               [default: 53]
  mp.dps = 50                 [default: 15]
  mp.trap_complex = False     [default: False]
```

mpmath has a nice function for square roots. Compare this to your approximation from before!

```
In [28]: sqrt(2)  # mpf(...) stands for an mp-float.
```

```
Out[28]: mpf('1.4142135623730950488016887242096980785696718753769468')
```

```
In [27]: type(sqrt(2)) # mpmath stores numbers in its own types!
```

```
Out[27]: mpmath.ctx_mp_python.mpf
```

```
In [26]: 4*mp.atan(1) # mpmath has the arctan built in.  This should be pretty close to pi!
```

```
Out[26]: mpf('3.1415926535897932384626433832795028841971693993751068')
```

Now try Archimedes' approximation of pi. Use the mpmath sqrt function along the way. How many iterations do you need to get pi correct to 100 digits? Compare this to the arctan-series (not the mpmath atan function) via Machin's formula.

```
In [23]: #  Explore and experiment.
         from math import pi
         from mpmath import *

         small_p = sqrt(8)
         big_P = 4

         mp.prec = 100
         mp.dps = 100

         for x in range(1000):
             big_P = 2/((1/small_p)+(1/big_P))
             small_p = sqrt(small_p*big_P)
             pi_apprx = (big_P+small_p)/2
         print(pi_apprx)
         print(abs((pi_apprx-pi)))
```

```
3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803482534
2.85746847820568745553945887076335239862977498705699665303781460806908158976684548918862005465
```

Answer:
We need 1000 iterations to reach a precision of 100 digits of pi. The Machin formula doesn't seem to be able to reach 100 digits of precision. This may be because of the data types.