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.

Table of Contents

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 (https://docs.python.org/3/library/stdtypes.html). The official Python tutorial (https://docs.python.org/3/tutorial/introduction.html) 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 [1]: 2 + 3
```

```
Out[1]: 5
```

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

```
Out[2]: 6
```

```
In [3]: 5 - 11
```

```
Out[3]: -6
```

In [4]:
```
5.0 - 11
```
Out[4]:
```
-6.0
```

In [5]:
```
5 / 11
```
Out[5]:
```
0.45454545454545453
```

In [6]:
```
6 / 3
```
Out[6]:
```
2.0
```

In [7]:
```
5 // 11
```
Out[7]:
```
0
```

In [8]:
```
6 // 3
```
Out[8]:
```
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 (https://en.wikipedia.org/wiki/IEEE_754) 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 [1]:
```
-12 // 5  # What will this output?  Guess before evaluating!
```
Out[1]:
```
-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 [2]:
```
Out[4] + Out[5]
```
```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-2-6ecae2ba42e6> in <module>()
----> 1 Out[4] + Out[5]

KeyError: 4
```

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 [11]:
```
(3 + 4) * 5
```

Out[11]:
```
35
```

In [12]:
```
3 + (4 * 5)
```

Out[12]:
```
23
```

In [13]:
```
3 + 4 * 5   #  What do you think will be the result?  Remember PEMDAS?
```

Out[13]:
```
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 [22]:
```
#  Practice cell
2+48347783*3456789

98977887/435
```

Out[22]:
```
227535.3724137931
```

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

In [23]:
```
2**1000
```

Out[23]:
```
10715086071862673209484250490600018105614048117055336074437503883703510511249361
22493198378815695858127594672917553146825187145285692314043598457757469857480393
45677748242309854210746050623711418779541821530464749835819412673987675591655439
46077062914571196477686542167660429831652624386837205668069376
```

In [24]:
```
2.0**1000
```

Out[24]:
```
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 [5]:
```
type(2**1000)
```

Out[5]:
```
int
```

In [6]:
```
type(2.0**1000)
```

Out[6]:
```
float
```

In [ ]:
```
#  An empty cell.  Have fun!
type(2**5890)
```

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.

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. Tuples are used for sets of data. Int is used for integers. Float is used for numbers with decimals. Strings are used for anything typed between quotes.
2. Division can have integers put into it but put out a float. When a float and int is inputed, then the output is a float. When two floats are inputed, then the answer is a float.
3. Multiplication follws the rules of PEMDAS, where multiplication comes before addition and subtraction.
4. 100 is an int in python, while 100.0 is a float.

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 (https://docs.python.org/3/library/stdtypes.html#boolean-operations-and-or-not)](https://docs.python.org/3/library/stdtypes.html#boolean-operations-and-or-not).

In [3]:
```
3 > 2
5.0/5.0
```

Out[3]:
```
1.0
```

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

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

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

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

In [ ]:
```
32 >= 31
```

In [ ]:
```
2 + 2 == 4
```

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

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

#23**32 is bigger than 32**23
```

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 [1]:
```
101 % 2 == 0  # Is 101 even?
```

Out[1]:
```
False
```

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

In [2]:
```
1234567890%3# Your code goes here.
#1234567890 is divisible by 3.
```

```
  File "<ipython-input-2-d433d6ea824a>", line 2
    1234567890 is divisible by 3.
                               ^
SyntaxError: invalid syntax
```

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 [3]:
```
True and False
```

Out[3]:
```
False
```

In [4]:
```
True or False
```

Out[4]:
```
True
```

In [5]:
```
True or True
```

Out[5]:
```
True
```

In [6]:
```
not True
```

Out[6]:
```
False
```

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

In [17]:
```
(2 > 3) and (3 > 2)
```

Out[17]:
```
False
```

In [18]:
```
(1 + 1 == 2) or (1 + 1 == 3)
```

Out[18]: True

In [19]:
```
not (-1 + 1 >= 0)
```

Out[19]: False

In [20]:
```
2 + 2 == 4
```

Out[20]: True

In [21]:
```
2 + 2 != 4  # For "not equal", Python uses the operation `!=`.
```

Out[21]: False

In [22]:
```
2 + 2 != 5  # Is 2+2 *not* equal to 5?
```

Out[22]: True

In [23]:
```
not (2 + 2 == 5)  # The same as above, but a bit longer to write.
```

Out[23]: True

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

In [16]:
```
# Experiment here.
not not not 2>1
```

Out[16]: 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 [9]:
```
False * 100
True*100
```

Out[9]: 100

In [13]:
```
True + 13
False+13
True+56
False and False or not True
```

Out[13]: False

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 (https://www.python.org/dev/peps/pep-0020/) 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.

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`?

Solutions

(Edit here to give solutions to the exercises)

1. True and True #True True and False #False False and True #False False and False #False True or True #True True or False #True False or True #True False or False #False
2. I might remember the truth tables by using mathematical logic. "And" and "or" do resemble the standard english usage by resmbling the english defintions of "and" and "or" and represent the answer with true or false.
3. (2349872348723%7=0) and not(2349872348723%11=0)
4. The order of operations is "not", then "and", then "or".
5. (not(True and True) and not(False and False))

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 [ ]:
```
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 [36]:
```
e = 2.71828
e * e
```

Out[36]:
```
7.3890461584
```

In [37]:
```
e = 2.71828
type(e)
```

Out[37]:
```
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 (https://google.github.io/styleguide/pyguide.html#Python_Style_Rules) for Python code.

In [ ]:
```
my_number = 17
```

In [ ]:
```
my_number < 23
```

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 [ ]:
```
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 [1]:
```
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 [8]:
```
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 [9]:
```
print(my_number)
print(new_number)
True==False
```

```
3.14
18
```

Out[9]:
```
False
```

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.)

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?

Solutions

(Use this space to work on the exercises.)

1. "=" is used to assign variable, while "==" is used for operations.
2. 9
3. a=c a=b b=c

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 [43]: type([1,2,3])
```

```
Out[43]: list
```

```
In [46]: type(['Hello',17])
```

```
Out[46]: list
```

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

```
In [45]: type((1,2,3))
```

```
Out[45]: 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 [44]:
```
type(range(10)) # Ranges are their own type, in Python 3.x.  Not in Python 2.x!
```

Out[44]:
```
range
```

In [14]:
```
list(range(10)) # Let's see what's in the range.  Note it starts at zero!  Where do
es it stop?
```

Out[14]:
```
[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 [15]:
```
list(range(3,10))
```

Out[15]:
```
[3, 4, 5, 6, 7, 8, 9]
```

In [48]:
```
list(range(-4,5))
```

Out[48]:
```
[-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 [49]:
```
len([2,4,6])
```

Out[49]:
```
3
```

In [18]:
```
list(range(10)) # The len command can deal with lists and ranges.  No need to conve
rt.
```

Out[18]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In [4]:
```
len(list(range(10,100))) # Can you figure out the length, before evaluating?
```

Out[4]:
```
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 [16]:
```
list(range(1,10,2))
```

Out[16]:
```
[1, 3, 5, 7, 9]
```

In [5]:
```
list(range(11,30,2))
```

Out[5]:
```
[11, 13, 15, 17, 19, 21, 23, 25, 27, 29]
```

In [10]:
```
list(range(-4,5,3))
```

Out[10]:
```
[-4, -1, 2]
```

In [16]:
```
list(range(10,100,17))
```

Out[16]:
```
[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 [53]:
```
list(range(10,0,-1))
```

Out[53]:
```
[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 [17]:
```
list(range(10,100,7))  # What list will this create?  It won't answer the question.
..
```

Out[17]:
```
[10, 17, 24, 31, 38, 45, 52, 59, 66, 73, 80, 87, 94]
```

In [18]:
```
list(range(14,100,7))  # Starting at 14 gives the multiples of 7.
```

Out[18]:
```
[14, 21, 28, 35, 42, 49, 56, 63, 70, 77, 84, 91, 98]
```

In [19]:
```
len(range(14,100,7))  # Gives the length of the list, and answers the question!
```

Out[19]:
```
13
```

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 [9]:
```
#  Use this space to work on the exercises.
#1
len(range(2,102))==(102-2)

#len(range(a,b))==(b-a)
#2
list(range(1,12))
#3
5*list(range(1,6))
#4
2700%3==0
len(range(300,3000,3))#900
```

Out[9]:
```
[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]
```

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 [57]:
```
for n in [1,2,3,4,5]:
    print(n*n)
```

```
1
4
9
16
25
```

In [58]:
```
for s in ['I','Am','Python']:
    print(s + "!")
```

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

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

In [59]:
```
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 [60]:
```
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 [29]:
```
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 [30]:
```
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 [31]:
```
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 [32]:
```
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 [33]:
```
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 [61]:
```
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 [62]:
```
# 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.

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 [34]:
```
#1
-2/2  # -1
-2/-2 #  1
#Python follows the normal laws of math, when the divisor or divided is negative.
#2
(2**90)%91 # The answer is 64
#3
a=1000//13  #76
#4
a/2  #38
#5
for n in range(1,1000):
    a=a+n
#6
print(a)
36
for n in range(1,1000):
    a=a+n**2
print(a)
```

```
499576
333333076
```

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.

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 [62]:
```
#  Start your explorations here!
guess=567890/2
for i in range(100):
    closer_square_root=0.5 * (guess + 567890/guess)
    guess=closer_square_root
    print(closer_square_root)
```

```
141973.5
70988.74998591286
35498.37485208983
17757.186242925494
8894.58354900659
4479.215135325703
2302.9992538001493
1274.7931970267111
860.1346870615723
760.1842476287068
753.6134127462518
753.5847668448372
753.584766300381
753.584766300381
753.584766300381
753.584766300381
753.584766300381
753.584766300381
753.584766300381
753.584766300381
753.584766300381
753.584766300381
753.584766300381
753.584766300381
753.584766300381
753.584766300381
753.584766300381
753.584766300381
753.584766300381
753.584766300381
753.584766300381
753.584766300381
753.584766300381
753.584766300381
753.584766300381
753.584766300381
753.584766300381
753.584766300381
753.584766300381
753.584766300381
753.584766300381
753.584766300381
753.584766300381
753.584766300381
753.584766300381
753.584766300381
753.584766300381
753.584766300381
753.584766300381
753.584766300381
753.584766300381
753.584766300381
753.584766300381
753.584766300381
753.584766300381
753.584766300381
753.584766300381
753.584766300381
753.584766300381
753.584766300381
753.584766300381
753.584766300381
753.584766300381
753.584766300381
```

The closer the starting guess, the closer the function will get to the actual square root. Changing the function to start with a guess of dividing by the original number by 2 will improve the speed of converging to the square root.

After the 5th iteration with the original number divided by 2, the algorithm converges to the square root of 2, but it takes 8 iterations for the algorithm to converge to the square root of 2 with the starting guess of 10.

With larger positive numbers, it takes longer for the algorithm to converge to the square root even when starting with the original number divided by 2.

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 [85]:
```
#  Approximate e here.

series=0
x=1
fact=1
for i in range(1000):
    for n in range(1,i+1):
       fact=fact*n
    seriesterm=(x**(i))/(fact)
    series= series+seriesterm
    fact=1
print(series)
```

```
2.7182818284590455
```

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 [15]:
```
#  Approximate pi here!
#1

x=.5
series=0
for n in range(100):
    seriesterm=(-x**2)**n
    series= series+seriesterm
print(series)

#2
x=1
series=0
for n in range(100000):
    seriesterm=(-1)**n*(((x)**(2*n+1))/(2*n+1))
    series= series+seriesterm
print(series*4)
```

```
0.8
3.1415826535897198
```

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 [30]:
```
y=1/239
series1=0
for i in range(10):
    seriesterm1=(-1)**i*(((y)**(2*i+1))/(2*i+1))
    series1= series1+seriesterm1
print(series1)
```

```
0.0041840760020747225
```

In [34]:
```
#  Approximate pi more quickly here!
x=1/5
series=0
for n in range(100000):
    seriesterm=(-1)**n*(((x)**(2*n+1))/(2*n+1))
    series= series+seriesterm
print(series)
y=1/239
series1=0
for i in range(10):
    seriesterm1=(-1)**i*(((y)**(2*i+1))/(2*i+1))
    series1= series1+seriesterm1
print(series1)
pi=4*(4*series-series1)
print(pi)
```

```
0.1973955598498808
0.0041840760020747225
3.141592653589794
```

Now let's compare this to **Archimedes' method**. Archimedes approximated pi by looking at the perimeters p(n) and P(n) of (2^n)-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/ (http://mpmath.org/doc/current/)

First we load the package and print its status.

In [7]:
```
from mpmath import *
print(mp)

#1
#p(2)=4*sqrt(2)
#P(2)=8
#2
psmall=4*sqrt(2)
pbig=8
pi=0
for t in range(10):
    pbig=(2*psmall*pbig)/(psmall+pbig)
    psmall=sqrt(psmall*pbig)
    pi=(pbig+psmall)/4
print(pbig,psmall)
print(pi)
```

```
Mpmath settings:
  mp.prec = 336               [default: 53]
  mp.dps = 100                [default: 15]
  mp.trap_complex = False     [default: False]
6.283186539258614621578917573655951528712864759942355480782713179525038360138156
641218306613268407385 6.283184691140235484680751988314739860610412151302407616056616615704913224827602310740136482118971252
3.141592807599712526564917390492672847330819227811190774209832448807487896241439737989610773846844659
```

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 [144]:
```
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 [145]:
```
sqrt(2)  # mpf(...) stands for an mp-float.
```

Out[145]:
```
mpf('1.4142135623730950488016887242096980785696718753769468')
```

In [146]:
```
type(sqrt(2)) # mpmath stores numbers in its own types!
```

Out[146]:
```
mpmath.ctx_mp_python.mpf
```

In [147]:
```
4*mp.atan(1) # mpmath has the arctan built in.  This should be pretty close to pi!
```

Out[147]:
```
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 [31]:
```
#  Explore and experiment.
from mpmath import *

mp.dps = 100

psmall=4*sqrt(2)
pbig=8
pi=0
for t in range(164):
    pbig=(2*psmall*pbig)/(psmall+pbig)
    psmall=sqrt(psmall*pbig)
pi=(pbig+psmall)/4
print(pi)

mp.dps = 100
x=mpf(1.0)/5

series=0
for n in range(70):
    seriesterm=(-1)**n*(((x)**(2*n+1))/(2*n+1))
    series= series+seriesterm
#print(series)
y=mpf(1.0)/239


series1=0
for i in range(70):
    seriesterm1=(-1)**i*(((y)**(2*i+1))/(2*i+1))
    series1= series1+seriesterm1
#print(series1)
pi=4*(4*series-series1)
print(pi)

#70 iterations is enough for Machin's formula, and 164 is enough for Archimedes pri
nciple.
```

```
3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803482534211706 8
3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803482534211706 8
```

In [ ]:

In [ ]: