Python basics

https://people.sc.fsu.edu/~jburkardt/classes/math1800_2023/python_basics/python_basics.pdf

---

Python Fundamentals

- *Python can be used interactively, as a calculator;*
- *We can save useful data using named variables;*
- *Number data includes integers, real numbers, and strings;*
- *There are some simple rules for choosing variable names;*
- *The `math` library includes most useful mathematical functions;*
- *Comparison operators return a True/False or "bool" value;*

1 Starting the Interactive calculator

If you want to access Python directly on your computer, you typically have to find an open a shell or terminal window. On a PC this might show up as an Anaconda PowerShell. The shell expects you to type commands from the keyboard. The command to start a Python session is

```
python
```

You may notice that I often start my session by typing `python3`. That's just a habit I got into, because it tells the operating system that I either want version 3 of Python, or nothing. Since Python version 2 is dead, my habit isn't really needed any more.

2 Interactive calculator

Once you have started a Python session, a simple process is to use it as a calculator. You can write formulas involving numbers, the operators +, -, *, /, //, **, %, and parentheses. Depending on how you are using it, Python will print out the value, or you may have to use the `print()` statement to examine your result.

```
>>> 3 + 5 * 4
23
>>> 365/12
30.416666666666668   # typical 16 digit output for real numbers
>>> 9**3             # 9 cubed
729
>>> 37 / 5           # division with real result
7.4
>>> 37 // 5          # division rounded down to integer result
```

```
7
>>> 37 % 5            # remainder for 37/5
2
>>> 4 * 3 + 2
14
>>> 4 * ( 3 + 2 )  # parentheses enforce a particular order
20
```

The >>> sign is the Python prompt, which indicates that Python is running interactively with you, and is waiting for your next command. We will include this sign for a while in this document, but after a while it can be distracting, and so we will gradually forget about it.

Notice the statements that begin with a hash mark. These are Python comments. You can include them in a Python computation to provide an explanation for what you are doing. In homework submissions, you can use comment statements to specify your name, and the homework problem you are working on.

If you have a very large or very small decimal value, you can use exponential format. Thus, 1200 can be written as 1.2E3, and 0.00345 as 3.45E-3.

Occasionally, you may have to deal with an integer with many digits. Ordinarily, to keep from going crazy and to help spot typos, you use commas to group the digits into threes. This is NOT acceptable to Python; the commas will make it misunderstand your input. However, you can instead use underscores where commas would ordinarily go. Suppose, therefore, that we want to find the square root of 1,524,157,875,019,052,100. Here is how to do it in Python:

```
  from math import sqrt
  n1 = 1,524,157,875,019,052,100  # Rejected, with interesting error message
  n2 = 1524157875019052100
  s2 = sqrt ( n2 )
  n3 = 1_524_157_875_019_052_100
  s3 = sqrt ( n3 )
```

It looks like Python was almost prepared to accept our answer with commas. Let's see what happens with a simpler example.

```
>>> x = 12,345,678
(12, 345, 678 )
type ( x )
<class 'tuple'>
```

Our commas were interpreted as separators of 3 items, to be packed into a single object called a *tuple*. We will talk about them more, later. But can you guess that happens if I type

```
x[1]
```

In some computations, it is necessary to use complex numbers. Python writes a complex number in the form a+bj or (a+bj), where j represents the imaginary unit, that is, $\sqrt{-1}$. Thus, we can write

```
>>> x = 2 + 3j
>>> y = -4j
>>> z = x * y
(12-8j)
```

Note that if you want to represent $\sqrt{-1}$, you write 1j, not just j:

```
>>> u = 1j
>>> v = u**2
(-1+0j)
```

We have just seen three kinds of numeric data: integers, reals, and complex values. Python classifies these as of type `int`, `float` and `complex` respectively. Usually, you don't need to worry about the types of your data, but there are some cases where Python will refuse to work with a value because it does not have the appropriate type. Python automatically "guesses" the type of your numeric data when you enter your statements, based on simple hints. A number is assumed to be an integer or float depending on whether a decimal point appears in its definition. A number is assumed complex if it has an imaginary part. Occasionally, it will be necessary to correct Python's assumption about the type of a quantity, and we will see how to do this later. If you have any doubts about how Python is viewing a variable, you can use the `type()` command to find out:

```
a = 1
b = 2.0
c = 3 + 4j
d = b / a
e = c.imag()
type ( abs )
```

3 Killing the Interactive calculator

If you are running Python in an interactive shell, then you can terminate the Python session by typing `quit()` or `exit`. The Python prompt `>>>` disappears, and you are back to talking to your computer's shell.

4 Variables

Having an interactive calculator can be useful, but Python can do much more for you. In particular, if we want to process data, we will need to get used to the idea of storing values in *variables*. A variable is simply a name to which we can associate information. We assign a value to a variable by using the equals sign.

For example, a hospital visit often begins by measuring the patient's height and weight. Rather than simply write these down as raw numbers, we can store them using variable names that make it easy to recall later.

```
  >>> weight_lb = 145
  >>> height_in = 60
```

The name that we give a variable must follow a few rules:

- formed from letters, digits, and underscores;
- cannot start with a digit;
- are case sensitive.
- are not allowed to be the same as certain Python reserved words, such as `and`.
- are allowed to be the same as some Python words, such as `sqrt`;

To see what happens with name conflicts, try

```
  2times4 = 2 * 4
  and = 6
  sqrt = 19
```

Once we have a named variable, we can use it on the right hand side of assignment statements, where it now is equivalent to the value we stored there. For instance, the statement

```
  >>> bmi = weight_lb / height_in**2 * 703
```

is equivalent to, but much easier to understand than,:

```
  >>> bmi = 145 / 60**2 * 703
```

Since we did not use parentheses in the expression on the right hand side, Python follows a set of rules for how to decide what is meant. In this case, the multiplication and division operators have the same importance (called "priority"), and so the expression is evaluated by handling the operators from left to right.

```
>>> bmi = ( 145 / ( 60**2 ) ) * 703     # what Python does
>>> bmi = ( 145 / 60 )**2 * 703 )       # what Python does NOT do
>>> bmi = 145 / ( ( 60**2 ) * 703 )     # what Python does NOT do
```

How would you compute powers of -1? What happens if you type

```
-1**2
```

A common mistake in writing Python expressions occurs when trying to represent a fraction, such as $y = \frac{x^2-1}{x-1}$. Parentheses are essential here. It would be quite wrong to write

```
>>> y = x**2 - 1 / x - 1    # This is NOT what we meant!
```

We should say something like

```
>>> y = ( x**2 - 1 ) / ( x - 1 )
```

Similarly, as we saw in the quadratic formula example earlier, the mathematical expression $\frac{-b+\sqrt{b^2-4ac}}{2a}$ must be carefully transcribed as

```
r1 = ( - b + sqrt ( b**2 - 4 * a * c ) / ( 2 * a )
```

The advantage of using variable names is that we can apply formulas to the symbolic names in a natural way. For instance, to convert from pounds to kilograms, we divide by 2.204. To convert inches to meters, we divide by 39.370. Thus, if our hospital actually uses the metric system, we can convert our readings using the following formulas:

```
>>> weight_kg = weight_lb / 2.204
65.789
>>> height_m = height_in / 39.370
1.524
```

and it turns out that the formula for BMI is simpler when we use metric units:

```
>>> bmi = weight_kg / height_m**2
```

5 The `math` library

A calculator has a few special function keys for square root, inverse, and maybe some trig functions. In scientific programming, there are many mathetmatical functions that we need to access. There are several such functions that are always available:

- `abs(x)` returns the absolute value of an integer, float, or complex number;
- `max(x,y)` returns the maximum of two integers or floats;
- `min(x,y)` returns the minimum of two integers or floats;
- `round(x)` rounds a float to an integer, rounding down (towards $-\infty$;

Most of the useful functions are reserved in the Python `math` library. One member of this library is the `sqrt()` function, which returns the square root of a float or integer. To be allowed to use this function, we have to "check it out" of the library, using an `import` statement. Here is an example in which we want to use the quadratic formula to determine the roots of the polynomal $p(x) = ax^2 + bx + c$:

```
from math import sqrt
a = 1
b = 1
c = -12
root1 = ( - b + sqrt ( b**2 - 4 * a * c ) ) / ( 2 * a )   # Need parentheses top and
    bottom
root2 = ( - b - sqrt ( b**2 - 4 * a * c ) ) /   2 / a       # An alternative way to
    divide by 2*a
```

Here are some of the other functions in the math library. You can get a full list with the search term "python math":

- `ceil(x)`, `floor(x)`: rounds a float to an integer, rounding away from or towards from 0;
- `comb(n,k)`: the "n-choose-k" combinatorial function;
- `exp(x)`, `log(x)`: exponential and logarithmic functions;
- `factorial(n)`: n!;
- `gcd(m,n)`: greatest common divisor of m and n;
- `log2(x)`, `log10(x)`, logarithms base 2 and 10;
- `sqrt(x)`: $\sqrt{x}$;
- `sin(x)`, `cos(x)`, `tan(x)`, `sec(x)`, `csc(x)`, `cot(x)`;
- `asin(x)`, `acos(x)`, `atan(x)`;
- `sinh(x)`, `cosh(x)`, `tanh(x)`, `sech(x)`, `csch(x)`, `coth(x)`;
- `asinh(x)`, `acosh(x)`, `atanh(x)`;
- `erf(x)`, the "error" function;
- `pi`, `e`, the values of π and Euler's constant;
- `inf`, `nan`, the values of ∞ and "Not-a-Number";

For details about the Python math library, use Python's `help()` facility:

```
help ( 'math' )
```

6 A simple iteration

With our small amount of Python knowledge, we can nonetheless pose and solve some simple mathematical questions. Later we will see how to do these same operations in a much more elegant way, but for now it's important to get a feeling for how such calculations are carried out.

Consider the question of whether there is value of x for which $\cos(x) = x$? If you sketch plots of the functions $y = x$ and $y = \cos(x)$, you will see that there is an intersection in the unit interval. But suppose you want to get a good approximation to this value, rather than just guessing from a plot.

Let's start by checking out the `cos()` function from the `math` library. Now we will be doing a guessing game, and we have to give an initial guess, so we'll say $x = 1$.

```
>>> from math import cos
>>> x = 1.0
```

Now we are going to replace x by the value $\cos(x)$. Since we want to print this value automatically, we are going to tack on a `print()` statement on the same command line. That is possible as long as we separate the two statements by a semicolon:

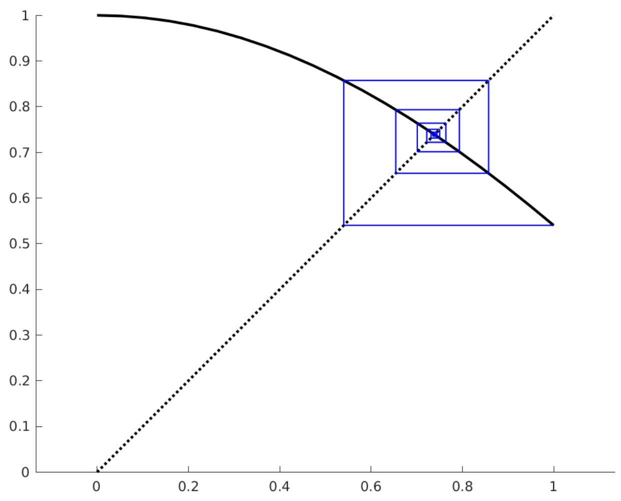
```
>>> x = cos ( x ); print ( x )
```

Now the funny thing is that we can repeat this command simply by hitting the up-arrow key. And the even funnier thing is that the values we compute seem to be converging. Here are the first 10 values I computed:

```
 0  1.0
 1  0.5403023058681398
 2  0.8575532158463934
 3  0.6542897904977791
 4  0.7934803587425656
 5  0.7013687736227565
 6  0.7639596829006542
 7  0.7221024250267077
 8  0.7504177617637605
 9  0.7314040424225098
10  0.7442373549005569
```

So the first digit has stopped changing, and the second is about to settle down. If you are patient, you can determine the third and fourth digits, and so on.

To see what is going on with this iteration, we can make what is called a "cobweb plot". We draw two lines, $y = x$ and $y = \cos(x)$. We start at $x = 1$ and evaluate $y = \cos(x)$. Then we draw a horizontal from the cosine curve to the diagonal line. This essentially carries out the operation $x = y = \cos(x)$. Now we evaluate $y = \cos(x)$ at this new position, and repeat the process. In this case, the image suggests how the process is converging in a sort of spiral to a fixed point. Of course, a nice convergence behavior like this doesn't always happen!



If the iteration example interests you, you can investigate two more iteration formulas. One of them converges, while the other seems to be chaotic. Again, you can use a starting point $x = 1$.

```
x = x/2 + 1/x
x = 2 * abs ( x - 0.5 )
```

These two expressions are computational assignment statements; however, if the iteration converges, we are hoping to find an x for which they are also legitimate mathematical equalities. If you look at the first statement as a mathematical equality, then you can actually solve it to determine the two "magic" values of x.

You should have noticed that, in Python, the = sign does not really indicate equality, but rather *assigment.*

In assignment statements, some formula appears on the right hand side of an equals sign, and a variable name appears on the left, into which the value of the formula will be stored.

```
>>> a = 1
>>> b = a + 2
>>> c = a * b + 3
>>> a = a + 2
```

The last assignment in the above list would make no sense to a mathematician. But you should understand its meaning in Python: *Get the current value of* `a`*, add 2, and use this as the updated value of* `a`. In some programming languages, assignments are represented using a special character like `:=` or `<-` to try to be more clear.

7 Comparisons, Logical variables and Logical Operators

Sometimes we need to ask a question before we carry out an assignment. For instance, before evaluating the formula $y = \frac{a}{b}$, we might want to know whether the denominator b is zero. When we are carrying out an iteration and we can measure the error in our approximation, we might want to know whether our error is smaller than some tolerance. These are examples of situations in which we want to make a comparison and get a logical answer. Depending on our interest, there are six operators we can choose from when comparing two values x and y.

- `x == y` is True if x is equal to y, and False otherwise;
- `x != y` is True if x is not equal to y, and False otherwise;
- `x < y` is True if x is less than y, and False otherwise;
- `x <= y` is True if x is less than or equal to y, and False otherwise;
- `x > y` is True if x is greater than y, and False otherwise;
- `x >= y` is True if x is greater than or equal to y, and False otherwise;

If we wish, we can assign the value of a comparison to a variable. This variable will be of a new type, known as a *boolean variable*, with Python type `bool`. Boolean variables can only have one of two values, `True` or `False`, spelled exactly like that. We can use a boolean variable to remember the result of a comparison. As a simple example, we might want to know whether an integer is even or odd. We could write

```
even = ( ( n % 2 ) == 0 )     # You could leave out all the parentheses,
                              # but I don't recommend it!
```

We will see later, when we are ready to talk about `if()` statements, that the logical value of these comparison operators will be used to allow our programs to make decisions. That is, something like this:

```
if ( comparison is True ):
  do this
else:
  do that
```

In some cases, our comparisons will be a little more complicated. We may want to know whether two things are true (an `and` situation) or at least one of two conditions is true (an `or` situation). For instance, to ask whether the value x is in the interval $[a, b]$, we might use the following expression:

```
inside = ( ( a <= x ) and ( x <= b ) )  # Parentheses not necessary, but I prefer them
```

Python has an additional boolean operator, `not`, which reverses the value of a given logical expression. Thus, here are two ways of determining if the value x is outside the interval $[a, b]$,

```
outside = not inside       # assuming we already defined the value of ``inside''
outside = ( ( x < a ) or ( b < x ) )
```