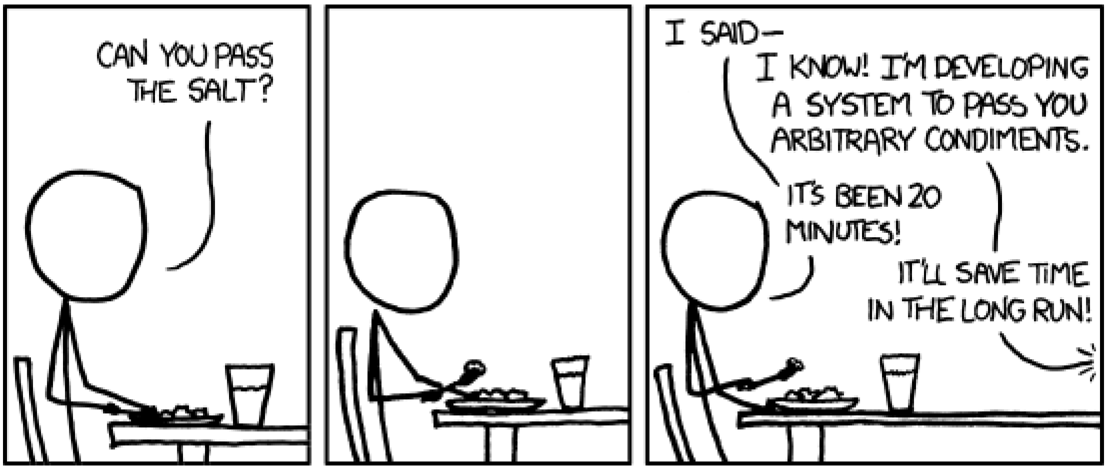
The basic grammar of Python


CAN YOU PASS THE SALT?
I SAID—
I KNOW! I'M DEVELOPING A SYSTEM TO PASS YOU ARBITRARY CONDIMENTS.
IT'S BEEN 20 MINUTES!
IT'LL SAVE TIME IN THE LONG RUN!

1. Variables and types

Variables

Variables are named entities that are stored in memory. For example, for integers *i* and *j*:

In [1]:

```
i = 0      # i is assigned the value 0 and then stored in memory
i          # display the value of i
```

Out[1]:

```
0
```

The pretty colours are called **syntax highlighting**; they help the programmer read and understand the code. Everything after the **hashtag #** is a **comment** and the program will not try to execute it. The `In[1]` statements are the code that is passed to python, and the `Out[1]` statements are what python returns, and numbered in sequence of execution of the code.

We can now start to **modify** our variable. If the following statement bothers you, think of the `=` not as a mathematical equal sign, but as an assignment: assign the value to the right of the `=` statement to the variable on the left of it:

In [2]:

```
i = i + 1  # add 1 to the value of i and store the result back in memory
i
```

Once we have one variable, we can use it to define others:

In [3]:

```
j = i    # Define a new variable j and assign the value of i to it
j
```

Note how we can continue do modify i without changing j - it retains the value that we gave it initially (we will later call this passing an argument by value).

In [4]:

```
i = i + 1   # add one more to the value of i
i           # the value of i has now changed in memory, it is 2 now
```

Out[4]:

```
2
```

In [5]:

```
j           # however, the value of j has not: it retains the value that we gave
 it originally
```

Out[5]:

```
1
```

Types of variables

We have already encountered two types of variables:

Integers, like `5, 6, 42, -2, 0`

Strings, like `'a is', 'python', 'i', str(5)`

Integers

are stored in the memory as either 32 bit or 64 bit binary numbers, in base 2, e.g.

```
10010 = 1*16 + 0*8 + 0*4 + 1*2 + 0*1 = 18
```

Another bit is added at the front for the sign.

Basic mathematics with integers is straightforward:

In [26]:

```
# define my i and j variables.
# With ; you can pute more than a statement onto one line (to be used sparingly)
i = 5; j = 3

# basic mathematical operations
a = i + j   # addition
b = i - j   # subtraction
c = i*j     # multiplication
d = i/j     # division
e = i**j    # taking an exponent
f = (i + j)/(i - j)   # brackets
```

You can add and subtract integers as usual. However, be careful with division: 4/2 gives the correct result,

In [6]:

```
int(4/2)
```

Out[6]:

```
2
```

however, 3/2 gives the result 1 on some machines (not this one though)!

In [7]:

```
int(3/2)
```

Out[7]:

```
1
```

To access the remainder of the integer division, use the % operator:

In [8]:

```
3%2
```

Out[8]:

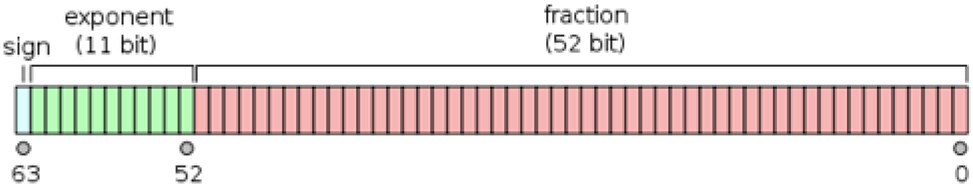
```
1
```

Floating point numbers

To work with **real numbers**, also called **floating point numbers**, we use floats (32 bit) or doubles (64 bit), for example

```
1.02, 3.141592, sqrt(2), 3.0/2.0, -12.3, 1.05457 10^(-34)
```

Doubles are stored with 1 bit for the sign, 11 bits for the exponent, and 52 bit for the fraction before the exponent.



Here are a couple of examples (we will talk about the `math` and other libraries later):

In [9]:

```
from math import *
sqrt(2)
```

Out[9]:

```
1.4142135623730951
```

In [10]:

```
pi
```

Out[10]:

```
3.141592653589793
```

In [11]:

```
1.05457*10**(-34)   # note formatting of the exponent using **
```

Out[11]:

```
1.05457e-34
```

Booleans

are `True` or `False` statements, and can be represented by a single bit (in practice they are stored in 32 or 64 bit).

You can assign them directly, like

In [12]:

```
lunch = False
lunch
```

Out[12]:

```
False
```

or they can emerge out of evaluations, like

In [13]:

```
a = 1 > 0
a
```

Out[13]:

```
True
```

In [14]:

```
b = i < j
b
```

Out[14]:

```
False
```

Lists

are collections of other variables in a sequence. They are separated by commas and put between square brackets.

A list of integers:

In [15]:

```
alist = [1, 3, -1, 5]
alist
```

Out[15]:

```
[1, 3, -1, 5]
```

The numbers between 0 and 9 (not 10)

In [16]:

```
blist = range(10)
blist
```

Out[16]:

```
range(0, 10)
```

We can also make lists of doubles, or lists of booleans.

To access an element of a list, use an index or iterator:

In [17]:

```
alist[0]
```

Out[17]:

```
1
```

In [18]:

```
blist[7]
```

Out[18]:

```
7
```

You can also assign a value to an element in a list and so modify it:

In [19]:

```
alist[3] = 4
alist
```

Out[19]:

```
[1, 3, -1, 4]
```

If you have coded in Matlab before:

- Note that the elements of a list of length N start at 0 and end at $N-1$.
- Note the square brackets.
- Note that python lists cannot do mathematics. Numpy arrays can, and we will introduce them later.

Strings

are **lists of characters**, and each character, like `‘p’` is saved as an 8 bit encoding. We use the `UTF-8` , or ‘Universal Character Set + Transformation format – 8-bit’ standard.

This is what the cryptic preamble in your Spyder scripts means: `# -*- coding: utf-8 -*-`

You can manipulate strings like a list and extract individual elements:

In [20]:

```
c = 'python'
c[2]
```

Out[20]:

```
't'
```

We can stitch strings together into larger statements. This is known as string addition or string concatenation, and it works with a simple `+` :

In [21]:

```
course = 'This is ' + c  # what happens if I delete the trailing space?
course
```

Out[21]:

```
'This is python'
```

If we want to know the value of a variable, we can use a `print()` statement to **print it to output**. That will convert anything (within reason) to a string and show its value.

In [22]:

```
print(course)
print(pi)
print('pi')
print(2+2)
```

```
This is python
3.141592653589793
pi
4
```