SEE1002
Introduction to Computing for Energy and Environment

Part 2: Elements of Python programming

Sec. 1: Data and variables

Course Outline

Part 1: Introduction to computing

Part 2: Elements of Python programming

Section 1: Data and variables

Section 2: Elementary data structures

Section 3: Branching or decision making

Section 4: Loops

Section 5: Functions

Part 3: Basic Python programming

Section 1: Modules

Section 2: Structure of a Python program

Section 3: Good programming practices

Part 4: Python for science and engineering

Section 1: File input and output

Section 2: NumPy and SciPy

Objectives

1. Understand how to store information in variables.
2. Understand how to operate on variables.

Outline

1. Basic data types
2. Basic operations

1. Basic data types

Overview

- We've already explained that computers can do only 2 things:

  1. Perform calculations ('operations')
  2. Remember results of these calculations/operations

- Hence the basic elements of any computer language will be concerned with these two points, i.e., storage of data and operations on data.

Introduction to data types

- Data simply refers to information or values.
- As far as the computer is concerned, everything consists of 0s and 1s. This is binary data.
- Humans prefer to classify data according to different data types.

Common data types

- The following data types are standard:
  - integer: -1, 0, 1, etc.
  - float: 3.1415, 2.7141,
  - string: “dog,” “cat”,“3 blind mice”,“Python is fun!”
  - Boolean: True or False

these are real numbers with decimal places

values are inside quotes

case is important!

- More complicated data types exist. We will talk about them later.

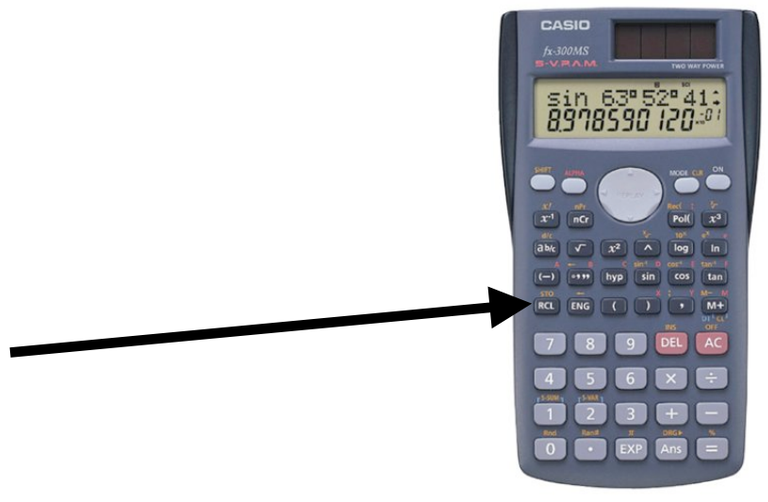
Variables

- It's convenient to assign a label or variable to specific values:

$$x = 1$$

$$y = 2$$

- This is what you learned in high school algebra. It's also a feature of most scientific calculators:



Unless you have a very fancy calculator, storage will be pretty limited (e.g. a single variable)

Use of variables in programs

- It's almost impossible to write a useful program without variables!
- We use variable to store information. Subsequently we can retrieve this information or perform operations.
- Examples
  - Student information for all of the students in this class.
  - Student id for a specific student
  - Average CGPA for the entire class

Key point about variables: **they are allowed to change**. This is very useful for solving problems.

How many variables do we need?

- If one is using simple equations (e.g. pV = nRT), we don't need many.
- But real engineering problems are more complicated than what you learned in high school:
  - Real-world equations can have lots of variables.
  - We need more variables if the physical quantities vary in space and time, e.g., T(x,t).

Scientific notation

- Scientific notation is convenient for humans:

$$3 \times 10^8, \quad 6.3 \times 10^6, \quad 6.02 \times 10^{23}$$

- For computers we need simper notation. A number $\texttt{ax10}^{\texttt{b}}$ can be represented as:

`aeb` or `aEb`

2. Assignment and retrieval

What is assignment?

Assignment stores information in a variable. More precisely, we *assign a value*.

Assigning numerical values

- The procedure follows elementary algebra. For an int:

variable ⟶ `In [1]: a=1` ⟵ value

- The same procedure, `variable=value`, also works for a float:

```
In [2]: x=3.14
```

`In[n]` denotes the nth line of input

What is retrieval?

Retrieval returns the information store in a variable. More precisely, we *retrieve a value*.

Retrieving numerical values

- In the Python shell we can retrieve the value by typing the name of the variable

```
In [3]: a
Out[3]: 1

In [4]: x
Out[4]: 3.14
```

Note that this only works in the shell. We will soon explain how to retrieve values inside of a Python program.

`Out[n]` denotes the nth line of output

Reassigning values

- We can modify a variable by assigning a new value to it:

```
In [1]: a=1

In [2]: x=3.14

In [3]: a
Out[3]: 1

In [4]: x
Out[4]: 3.14
```

original assignment

```
In [5]: a=2

In [6]: x=9.8
```

modification

```
In [7]: a
Out[7]: 2

In [8]: x
Out[8]: 9.8
```

Undefined variable

- Note that we cannot retrieve a value from an undefined variable:

```
In [24]: a=1

In [25]: a
Out[25]: 1

In [26]: b
Traceback (most recent call last):

  File "<ipython-input-26-89e6c98d9288>", line 1, in <module>
    b

NameError: name 'b' is not defined
```

Checking on a variable

- The shell allows us to check the value of a variable in several different ways. In addition to retrieval:

```
In [5]: a=2

In [6]: x=9.8
```

```
In [7]: a
Out[7]: 2

In [8]: x
Out[8]: 9.8
```

- We can also use the ? operator by typing it after the variable name

```
In [22]: a?
Type:        int
String form: 2
```

```
In [23]: x?
Type:        float
String form: 9.8
```

This provides us with more information. Note that Python uses `int` and `float` for the names of the data types.

ii) String values

- How do we distinguish generic variables from text?
- A string is arbitrary list of characters (e.g. letters, symbols and numbers).
- It's defined by enclosing their values within single quotes ' ' or double quotes " "

```
In [11]: dog='beagle'

In [12]: dog
Out[12]: 'beagle'
```

```
In [13]: cat="kitty"

In [14]: cat
Out[14]: 'kitty'
```

Single quotes vs. double quotes

- Should we define strings with single quotes or double quotes?
- In fact, either choice is fine.

```
In [20]: string1='dog'

In [21]: string2="dog"

In [22]: string1
Out[22]: 'dog'

In [23]: string2
Out[23]: 'dog'
```

```
In [24]: string1?
Type:        str
String form: dog
Length:      3

In [25]: string2?
Type:        str
String form: dog
Length:      3
```

string1 and string2 are identical

- Generally single quotes are preferred for short strings.

```
In [81]: shortvariable='tigger'

In [83]: longvariable="hunting for heffalumps is very dangerous"
```

What happens if we forget to include quotes?

- Usually we get an error message:

```
In [16]: mouse=Mickey
Traceback (most recent call last):

  File "<ipython-input-16-da9585c35446>", line 1, in <module>
    mouse=Mickey

NameError: name 'Mickey' is not defined
```

Missing quotes

```
In [17]: mouse="Mickey"

In [18]: mouse
Out[18]: 'Mickey'
```

With quotes

- Sometimes we may wish to assign the value of another variable:

```
In [19]: name='iphone'

In [20]: phone=name

In [21]: phone
Out[21]: 'iphone'
```

iii) Boolean variables

- Boolean variables are defined in the same way as before:

```
In [28]: weekday=True
```

```
In [29]: weekend=False
```

- It's conventional to associate `True` with 1 and `False` with 0.

Why are Boolean variables useful?

- In the mathematics and science courses you've taken, real numbers are used in all of the formulas. So why do we need Boolean variables?
- Remember that computers are only capable of distinguishing between 1 and 0.
- This means that Boolean variables play a very important role in computer programming.
- *Examples*
  - If a condition is true, do something
  - While a condition is true, continue do something

Naming variables

Python variables obey the following rules:

- No spaces or special characters (e.g. `+`, `-`, `/`, `*`, etc.)
- Case sensitive
- Numbers are fine
- Can't use reserved keywords (e.g. `print`)
- Variables can be very long, but long variable names are almost never used in practice.

Assigning multiple values

- Usually we do one assignment per line.
- Sometimes it's useful to do simultaneous assignment of multiple values:

```
In [2]: x,y=2,3

In [3]: x
Out[3]: 2

In [4]: y
Out[4]: 3
```

integer variables

```
In [5]: a,b,c = 2,'dog',3.1415

In [6]: a
Out[6]: 2

In [7]: b
Out[7]: 'dog'

In [8]: c
Out[8]: 3.1415
```

mixed variables

Automatic assignment

In some languages (e.g. Flowgorithm), we need to a declare a data type. That is, we need to tell the computer that a variable is an int or float, etc.

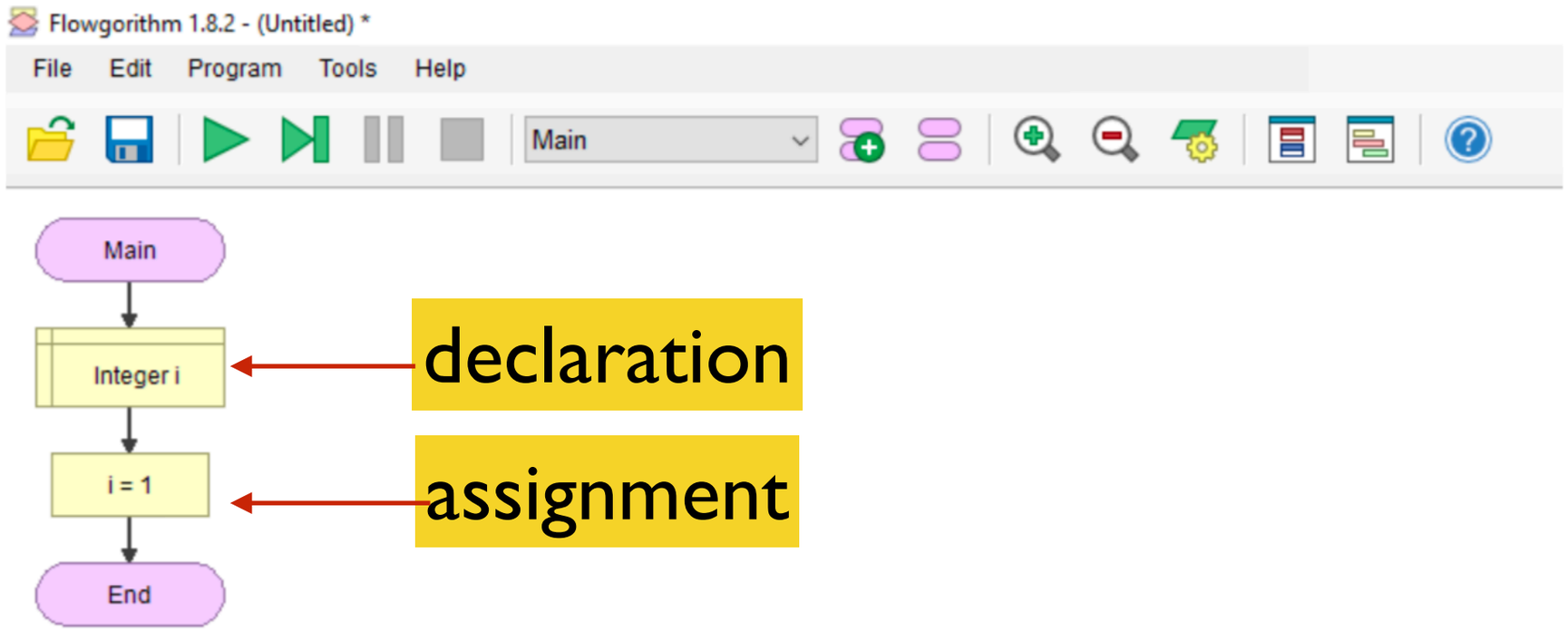
In Python, the data type is automatically determined based on the value. It usually guesses correctly. We can confirm this using the ? operator:

```
In [35]: a=1

In [36]: a?
Type:        int
```

```
In [37]: b=1.0

In [38]: b?
Type:        float
```

```
In [44]: schoolday=True

In [45]: schoolday?
Type:        bool
```

Variable declaration in Flowgorithm



Assignment by expression

- So far we've only introduced assignment by value. This is of somewhat limited use:

```
In [5]: V=17.5

In [6]: V
Out[6]: 17.5
```

value

- Most of the time we prefer to consider assignment by expression.

```
In [70]: I=3.5

In [71]: R=5.0

In [72]: V=I*R

In [73]: V
Out[73]: 17.5
```

expression

Typographical conventions

For ease of identification, **Python commands or statements** will be typeset in `Courier`.

Optional text (i.e. that must be input from the keyboard) will be enclosed inside angle brackets, e.g. `<filename>`.

3. Basic operations

Operations on variables

There are many, many things that we can do to variables or data. For simplicity, we can classify them as follows:

1. Data manipulation
2. Data comparison
3. Data input/output
4. *Data consolidation or organisation*

1. Data manipulation

- This is a fancy way of saying that we're interested in doing things to data.
- As a concrete example, we might collect data in an experiment or from a survey. In order to draw some useful conclusions from the data, we need to manipulate or process it in some way.

Types of operations

- How do we classify the operations that we can perform?
- For convenience, we distinguish between operations on numbers and operations on strings (e.g. letters + special characters).

a. Numeric operators

b. String operators

A. Numeric operators

- The most important are the basic arithmetic operations, i.e., +, -, *, / .
- Other examples include
  - Raising a number to a power: `2**3`
  - Remainder: `10%3`
  - Absolute value: `abs(-1)`

How do the numeric operators affect the data type?

- Numeric operators yield an output value. Is the output data type the same as the input data type?
- In most cases they preserve the type. But this isn't always the case…

Example 1: Identical data types

When the input values have identical data types, the numeric operators yield an output value of the same type.

```
In [16]: a=1

In [17]: b=2

In [18]: c=a+b

In [19]: c
Out[19]: 3

In [20]: c?
Type:        int
```

```
In [10]: a=1.0

In [11]: b=2.0

In [12]: c=a/b

In [13]: c
Out[13]: 0.5

In [14]: c?
Type:        float
```

This is usually ok. But occasionally there are surprises.

Example 2: integer division

Preserving the type is sometimes undesirable. The most common example is division of two integers. *Python automatically converts the output to a float.* This isn't done in all languages (e.g. Python 2).

```
In [4]: a=4

In [5]: b=2

In [6]: c=a/b

In [7]: c?
Type:        float
String form: 2.0
```

```
In [8]: a=1

In [9]: b=2

In [10]: c=a/b

In [11]: c
Out[11]: 0.5

In [12]: c?
Type:        float
String form: 0.5
```

Type conversion

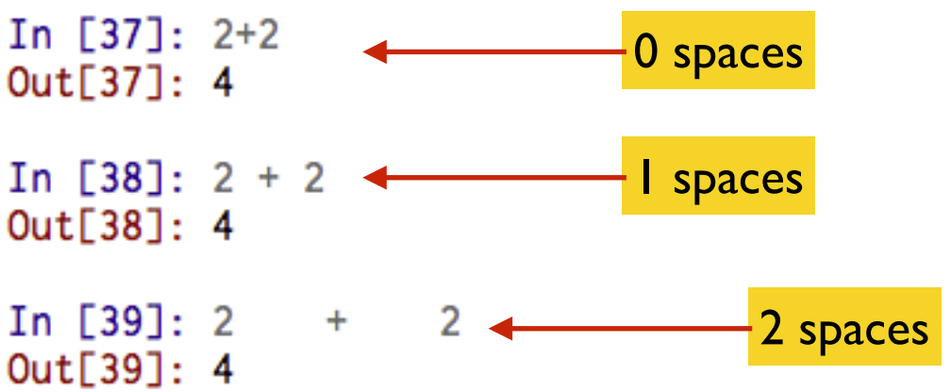
- In order to avoid surprises, we can change the data type. This is referred to as type conversion.
- This can be done with several operators:
  - `int()` Convert a float to an integer
  - `float()` Convert an integer to a float

```
In [28]: int(3.5)
Out[28]: 3
```

```
In [29]: float(3)
Out[29]: 3.0
```

Does whitespace matter?

- We can include as much space as we want between the input variables and the numeric operator.



- Generally it's good practice to skip spaces for binary arithmetic operators, but include them in an assignment.

```
a = 2+2
```

- However, adding spaces between terms can be helpful:

```
a = 2*(3+4) + (15**3.)/2.7
```

Summary of numeric operators

Operation	Operator	Comments
Addition	x + y	*x* and *y* may be floats or ints.
Subtraction	x - y	*x* and *y* may be floats or ints.
Multiplication	x * y	*x* and *y* may be floats or ints.
Division	x / y	*x* and *y* may be floats or ints. The result is always a float.
Remainder or Modulo	x % y	*x* and *y* must be ints. This is the remainder of dividing *x* by *y*.
Exponentiation	x ** y	*x* and *y* may be floats or ints. This is the result of raising *x* to the y^{th} power.
Float Conversion	float(x)	Converts the numeric value of *x* to a float.
Integer Conversion	int(x)	Converts the numeric value of *x* to an int. The decimal portion is truncated, not rounded.
Absolute Value	abs(x)	Gives the absolute value of x.
Round	round(x)	Rounds the float, *x*, to the nearest whole number. The result type is always an int.

From Lee, p.23.

B. String operators

- String variables are usually used for textual information. Examples:
  - first name
  - last name
  - degree programme
- In practice, strings can be manipulated in many ways…

Useful string operators

- concatenation: `combined = str1 + str2`
- length: `len(str)` is the number of characters
- indexing: `s[i]` is the ith character of the string `s`. The indexing starts at 0. For a range `min:max`, characters `min` to `max-1` are shown.

Example 3: Concatenation

We can form new string variables by combining them using +:

```
In [86]: var1='Homer'

In [87]: var2='Simpson'

In [88]: var1+var2
Out[88]: 'HomerSimpson'

In [89]: var1+' '+var2
Out[89]: 'Homer Simpson'
```

```
In [90]: var3=var1+var2

In [91]: var3?
Type:          str
String form: HomerSimpson
Length:        12
```

```
In [93]: var4=var1+' '+var2

In [94]: var4?
Type:          str
String form: Homer Simpson
Length:        13
```

Example 4: Determining the length of a string

It's easy to count the number of elements in a string using `len()`:

```
In [97]: var='qwertyuiop[]\asdfghjkl;'

In [98]: len(var)
Out[98]: 22
```

Example 5: indexing

We can form substrings by referring to specific character ranges.

```
In [99]: var1='Homer'

In [100]: var1[0]
Out[100]: 'H'

In [101]: var1[1]
Out[101]: 'o'

In [102]: var1[0:5]
Out[102]: 'Homer'
```

Summary of string operators

Operation	Operator	Comments
Indexing	s[x]	Yields the x^{th} character of the string *s*. The index is zero based, so s[0] is the first character.
Concatenation	s + t	Yields the juxtaposition of the strings *s* and *t*.
Length	len(s)	Yields the number of characters in *s*.
String Conversion	str(x)	Yields the string representation of the value of *x*. The value of *x* may be an int, float, or other type of value.
Integer Conversion	int(s)	Yields the integer value contained in the string *s*. If *s* does not contain an integer an error will occur.
Float Conversion	float(s)	Yields the float value contained in the string *s*. If *s* does not contain a float an error will occur.

We've already discussed type conversion

Fig. 1.19 String operations

From Lee, p.26

II. Data comparison

- Data manipulation is important but it can't be done all the time.
- We also need to compare data (this often precedes manipulation)
- This is done with relational and logical operators.

A. Relational operators

- Data can be compared with comparison or relational operators.
- We can just use the familiar relational operators from mathematics:

Python

mathematics

Operator	Condition
<	Less Than
>	Greater Than
≤	<= Less Than or Equal to
≥	>= Greater Than or Equal to
=	== Equal to
≠	!= Not Equal to

Lee, p.40

- Note that we use == for comparison because = is reserved for assignment.

Difference between = and ==

- A single = is used for assignment:

```
a=1
```

- We use == to test for equality:

```
In [29]: a==1
Out[29]: True
```

Output of relational operators

- As you'd expect, relational operators yield some kind of output.
- In fact, the output is very simple. The relational operators simply output `True` or `False`, i.e. a Boolean variable.

```
In [1]: x=1

In [2]: y=2

In [3]: x<y
Out[3]: True

In [4]:
```

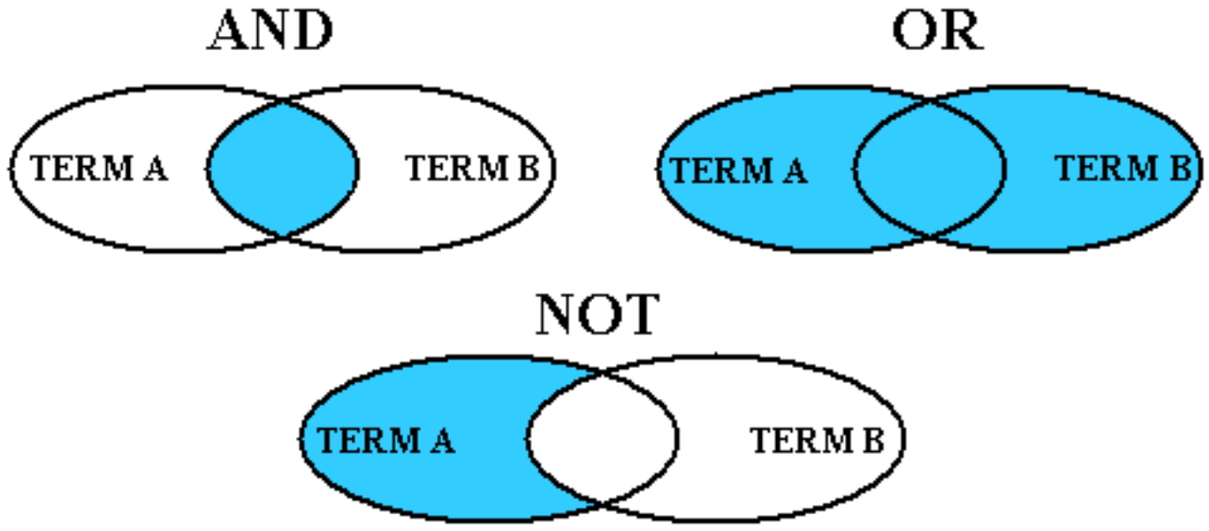
```
In [4]: a=1.0

In [5]: b=1.00000001

In [6]: a==b
Out[6]: False

In [7]: a!=b
Out[7]: True
```

```
In [8]: x>y
Out[8]: False
```

B. Logical operators

- We often need to determine whether multiple conditions hold.
- Examples:
  - First-year students must study calculus and physics
  - GE1301 or GE1331 will fulfil the major requirements.
  - SEE students must take SEE1002 not CS1101.
- These operations can be performed with logical operators.

Logical operators illustrated



link

Logical operators in Python

- The 3 most important logical operators are `and,or & not:`
  - `and`: True only if **both elements are True** (*applies to a pair of elements*)
  - `or`:True if **at least one element is True** (*applies to a pair of elements*)
  - `not`: **Changes state of the Boolean variable** (*applies to a single element*)

Example 7: logical operators

```
In [28]: weekday=True
```

```
In [29]: weekend=False
```

```
In [33]: not weekday
Out[33]: False

In [34]: not weekend
Out[34]: True
```

```
In [35]: weekday and weekend
Out[35]: False

In [36]: weekday or weekend
Out[36]: True
```

Evaluating statements

- The logical operators take Boolean variables, e.g. A and B, and return another one, i.e. True or False.
- But how do we know what we'll get?
  - and True only if **A and B are True**.
  - or True if **at least one of A and B are True**
  - not turns **True to False** and **False to True**.

Truth tables

- The behaviour of the logical operators is specified by truth tables.

A	B	A and B
False	False	False
False	True	False
True	False	False
True	True	True

Fig. 2.7 The and operator

A	B	A or B
False	False	False
False	True	True
True	False	True
True	True	True

Fig. 2.8 The or operator

A	not A
False	True
True	False

Fig. 2.9 The not operator

From Lee, p.59

Example 8: Logical operators

- Consider the following boolean variables:
  - `BEng = True`
  - `MSc = False`
- Use Python to answer the following questions:
  - Are `BEng and MSc` `True`?
  - Are `BEng or Msc True`?

III. Data input/output

- For obvious reasons, we need to get data into (input) and out of a variable (output).
- The approach we've adopted so far doesn't work well with real programs:
  - Referring to a variable, e.g. `variable <return>`, only generates output in the Python shell.
  - Assignments in the form `variable=value` aren't efficient.

A. Data output: `print`

- We can output data to screen or to a file.
- The `print` command can be used to output data to the screen. It's one of the most important commands in Python.
- Here's a famous example:

```
In [30]: print("Hello world!")
Hello world!
```

General form

A more general form of print is

```
print ( <expression> )
```

where *expression* is defined by variables or values separated by commas.

Later in the course we'll discuss formatted output. This allows us to output the data in a specified way (e.g. a certain number of decimal places, aligned in columns, etc.).

Typical uses of print

```
In [33]: a=1

In [34]: print(a)
1
```

single variable

```
In [38]: a=1

In [39]: b=2

In [40]: print(a,b)
1 2
```

multiple variables

```
In [43]: a=1

In [44]: print('a=',a)
a= 1
```

mixed output with a single variable

```
In [45]: a=1

In [46]: b=2

In [47]: print('a=',a,'b=',b)
a= 1 b= 2
```

mixed output with multiple variables

Example 9: basic use of print

```
In [55]: length=1.0

In [56]: width=2.0

In [57]: area=length*width

In [58]: print('The area is', area)
The area is 2.0
```

single variable

multiple variables

```
In [59]: print('length=',length,'width=',width)
length= 1.0 width= 2.0

In [60]: print('doubling the area=',2*area)
doubling the area= 4.0
```

expression

Comments

- `print` allows us to show a message on the screen for the user.
- Sometimes, however, we only want to leave a message for ourselves, i.e., the programmer.
- To do this we can leave a comment. In Python, comments start with #.

```
In [48]: # This is a comment

In [49]: x = 1 # define x

In [50]: x = 2*x # double x

In [51]: print(x) # output x
2
```

Why are comments useful?

- We don't really need comments when using the Python shell. In this case, the comment will always be seen!
- But comments are very useful when writing programs using an editor (e.g. Spyder):
  - They help explain the code to the programmer, i.e yourself or another person (e.g. the person marking your program!).
  - They are an easy way to temporarily disable part of a program.
  - Programmers often comment out code during testing.

We will illustrate the use of comments later in the course.

B. Data input

- We can also input data from the keyboard or a file. For now we're only going to discuss the former.
- Most of the programs that you've used require data to be input in some way from the keyboard.
- The commands that we're about to learn do essentially the same thing…

`input`

How do want to receive input from the keyboard?

- Ideally we want the computer to tell us to type something and then wait patiently for our response.
- In Python the `input` command command is the simplest way of doing this:

```
variable= input(str)
```

- This prompts the user to enter something with the keyboard, which is then stored in `variable`.
- `str` is an optional string or message
- `variable` is a string

Example 10: `input`

```
In [1]: number=input('Please enter a number: ')

Please enter a number: 3

In [2]: number
Out[2]: 3
```

inputting int

```
In [3]: number=input('Please enter a number: ')

Please enter a number: 12.3

In [4]: number
Out[4]: 12.3
```

inputting float

What about strings?

Since `input` has a string output, it also works with strings!

```
In [73]: x=input('Please enter something: ')

Please enter something: 1

In [74]: x?
Type:        str
String form: 1
Length:      1
```

inputting int

```
In [67]: x=input('Please enter something: ')

Please enter something: hello there!

In [68]: x
Out[68]: 'hello there!'

In [69]: x?
Type:        str
String form: hello there!
Length:      12
```

inputting string

Operating with `input`

Numerical data input from the keyboard must be converted.

```
In [81]: x=float(input('Please enter something: '))

Please enter something: 10

In [82]: x*x
Out[82]: 100.0
```

type conversion after input

```
In [83]: x=(input('Please enter something: '))

Please enter something: 100

In [84]: print(x*x)
Traceback (most recent call last):

  File "<ipython-input-84-16160cc886b9>", line 1, in <module
    print(x*x)

TypeError: can't multiply sequence by non-int of type 'str'
```

no type conversion after input

Note: using int() with a float input doesn't work.

Operating with `input` (2)

String data can be used directly.

```
In [85]: x=input('Please enter something: ')

Please enter something: hello

In [86]: print(x+x)
hellohello
```

no type conversion necessary

Summary

1. We can store information in memory by assigning a value to a variable.
2. There are number of distinct data types (e.g. integer, float, string).
3. Data can be compared with relational or logical operators.
4. Data can be input or output using `print` and `input`.