Introduction to Programming (SCMP142)

Ramses van Zon

October 2023

SciNet
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

Section 1

Introduction

SciNet
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

About this short course

The main point is to teach you the basics of programming!

We will be using the Python 3 programming language

Two one-hour sessions per week
Each sessions = short lecture + hands-on.

Topics

- Statements, expressions, variables, functions, objects
- Scripting
- Input and output
- Files and the file system
- Modularity

SciNet
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

Credit, certificates

- Completing this course counts for 8 credits towards a SciNet Certificate in Scientific Computing. (You would need 36 credits for the certificate.)
- Completing this course means:
  - Attending the sessions and taking the attendence test; and
  - Taking and passing the online test after the last session.
- Have to miss a session? Please inform us. In any case, you need to attend at least 5 sessions to get credit.
- This is *not* a course for UofT graduate credit.

What is Programming?

- We're going to get stuff done using a computer.
- But it will be stuff that is not be available in any existing application's menus.
- We will want to be able to repeat that same stuff again quickly.

SciNet
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

Required Software

Didn't work? Here's what you will need for the course:

- A Python installation

Make sure you get Python 3 (not Python 2).
(e.g. from https://www.anaconda.com)

The command to use Python 3 may be `python3` instead of `python`.

You could also work on SciNet:

```
$ ssh USERNAME@niagara.scinet.utoronto.ca
$ module load NiaEnv/2022a python/3.11.5
$ python
```

- A text editor

  You need an editor that can save in **plain text format**.
  (e..g., nano, emacs, vi, notepad, gedit, vscode, . . .)

  Working on an Apple device? Make sure you switch off " Smart quotes " in the settings.

- A terminal or command prompt

  Because running a Python program is easiest from the command line (a.k.a. "shell").

  E.g.: Bash, Mac Terminal, A-Shell, Anaconda Prompt, . . .

What is Programming?

- We're going to get stuff done using a computer.
- But it will be stuff that is not be available in any existing application's menus.
- We will want to be able to repeat that same stuff again quickly.

SciNet
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

Compute 9999 + 11111

Using a Graphical User Interface

- Start up your computer.
- On your computer, go to “Start”, “Applications”, “Calculator” (or your equivalent).
- Note: this should open a graphical calculator.
- On the keyboard, type `9999`.
- Then type `+`.
- Then type `11111`.
- Then type `=`.
- Read off the answer from the screen.
- Note: Alternatively, you can select the corresponding buttons on screen with mouse clicks.

Compute 9999 + 11111, again

Using the Python command line

- On your computer, open a "terminal".
- This should give you some form of a **terminal prompt** (or "shell prompt").
- `$` will be used to denote the terminal prompt in these slides, regardless of the form of the terminal prompt for your system.
- At the terminal prompt, type `python`.
- This will give you a message regarding the version of Python. (Try `python3` instead if the first number of the version is 2 or if just `python` does not work).
- It will also present you with a different looking prompt, the **Python prompt**, either `>>>` or `In[1]:`
- After the Python prompt, type `print(9999+11111)`, and press enter.

```
$ python
Python 3.9.12 (main, Apr  5 2022, 06:56:58)
[GCC 7.5.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or "license" f
or more information.
>>> print(9999+11111)
21110
>>>
```

Compute 9999 + 11111, and again

A first Python application

- On the same Python prompt, write

```
>>> f=open('app.py','w');f.write('print(9999+11111)');f.close()
```

- This creates a file called "app.py".
- That file contains just one line:

  `print(9999+11111)`

- Because it contains Python code, it is a Python application.
- To run the program, type `exit()` in Python, then type `python app.py`.
- Run the app again.

```
$ python
>>> print(9999+11111)
21110
>>> f=open('app.py','w')
>>> f.write('print(9999+11111)')
17
>>> f.close()
>>> exit()
$ python app.py
21110
```

```
$ python app.py
21110
```

Automation is what it's all about

- Automating the actions performed with a GUI is next to impossible.
- Once we had the text file "`app.py`", automation was easy.
- To create the file "`app.py`", requires some extra up-front work and knowledge (which we skipped over, and will usually do differently anyway)

Why No Integrated Development Environment?

- Although graphical, IDEs are not ideal for learning basic Python.
- The reason is that IDEs are big, with lots of configuration options and lots of functionality.

  When starting to program, that just distracts from the act of coding itself.

  Furthermore, every IDE is different, changes and evolves, so for instructional purposes, it is better to stick with what works for everyone, everywhere, anytime.

- *But the command line is so much harder!*

  It is just different, so there are some things to get used to.

  You may miss figuring out how to do something by looking at buttons and menus, pointing and clicking.

SciNet
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

How to figure things out in Python

1. You can get documentation for nearly any function and package.

```
>>> help(print)  # to get documentation about the print function
```

   Anything after "#" on a line is ignored by Python, it's a **comment** for your understanding.

   Someone told me/I read online that comments are unnecessary. **Don't listen to them.**

2. If you have the name of some data structure, you can ask what type it is.

```
>>> type(__name__)  # get the type, a.k.a. the class, of __name__
<class 'str'>
```

3. Given some data structure, look inside it.

```
>>> dir(__name__)  # look inside the __name__ structure
```

4. Or search the internet.

Hands-on #1: Installation Check

- Let's make sure we all have a working Python installation.
- We will be using Python 3.
- If you haven't installed Python yet, the easiest way to get it (currently) is probably anaconda.

- Open the terminal, type `python`, then, after the >>> prompt, type

```
print(9999+11111)
```

- This should cause the number 21110 to be printed on the screen.

The Python Ecosystem

- Although we will focus on the core Python language, the true strength of Python is the large body of available additional modules.
- These modules provide all kinds of functionality.
- There are many modules in the standard library that comes with Python ("batteries included").

  E.g. modules for GUIs, databases, random numbers, regular expressions, testing, ...

- There are even more third-party modules available.
- The official repository for third-party modules is the Python Package Index (https://pypi.org) with over 100,000 packages.
- Most Python distributions come with the "`pip`" command, with which you can install packages from pypi.

  (For Anaconda, you'd use the `conda` command instead).

Different interfaces to Python

There are a number of ways to use Python:

1. Standard, non-interactive mode of Python

   Open a terminal and type `python <SCRIPTNAME>`, and the code gets executed.

2. Standard, interactive mode of Python

   Open a terminal and type `python`, and you get a prompt like `>>>`.

   You can type commands at the prompt, they get executed, then you get another prompt.

3. IPython interactive mode

   Requires IPython installation. Then type `ipython`, and you get a `In [1]:` prompt.

   Has tab completion, command history, special commands.

4. Jupyter notebooks

   Input and output cells in your browser, with the Python back-end running possibly remotely. Harder to convert to scripts.

   https://jupyter.scinet.utoronto.ca

   a. Jupyter Notebook
   b. Jupyter Lab
   c. VS Code jupyter emulation

Which should I use?

- Personally, I would recommend **IPython** for interactive work during this course.
- Just keep in mind some of IPython's special commands will not work in pure Python scripts.
- IPython has a special command to save and reload your session:

```
In[1]: a='Hello'
In[2]: b='World'
In[3]: print(a,b)
Hello World
In[4]: %hist -f mysession.py
In[5]: %load mysession.py
```

- The slides will nonetheless have the regular Python prompt >>>
  and everything will work both in regular Python and IPython.
  (In contrast to $ which stands for the terminal prompt.)

SciNet
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

Section 2

What does Python really do?

Interpretation

What happens when we type `print(9999+11111)` on the Python prompt?

- First, note that Python was waiting for input, and allows you to edit that input. It doesn't 'do' anything until you hit enter.

  (in case of IPython, you can scroll through history and use tab completion, which are not 'doing nothing', but are still not doing Python)

- Once you hit enter, Python will check syntax, identifying functions, keywords, arguments, special characters, ...
- If it makes sense syntactically, it will then execute that command, i.e. translate it into (nested) function calls that at the lowest level are in machine code that the CPU understands.
- Python does this one line at a time, which puts it in the category of **interpreted languages**.

Example: 9999+11111

```
>>> print(9999+11111)
```

First action by Python: Syntax checking ("Parsing")

- `print` is a name.
- It should be a function, because it is followed by parentheses.
- The argument of the function is 9999+11111
- This is two 'literals' (numbers), separated by the plus sign, which is valid.

Second action by Python: Execution

- Store the integer 9999.
- Store the integer 11111.
- Call the + operator, with those integers as arguments.
- This "returns" a new integer.
- The print function is called with that new, temporary integer as an argument.
- Temporary integers are discarded.

Section 3

Basic elements of the Python language

Variables

- You can give names to values in Python

```
>>> firstnumber = 9
```

  We call this name-giving "assignment".

- You can reuse a name:

```
>>> firstnumber = 9999
```

- The earlier value of `firstnumber` no longer has that name anymore.

  Effectively, `firstnumber` has changed value.

- You can use variable instead of the value they refer too.

```
>>> print(9999)
9999
>>> print(firstnumber)
9999
```

- There are restrictions to the names: it can have letters, numbers, underscore, but cannot start with a number. No spaces, periods, brackets, etc., and they cannot be one of the reserved Python keywords:

```
and assert in del else raise from if
continue not pass finally while yield
is as break return elif except def
global import for or print lambda
with class try exec
```

Different types/classes of values in Python

- Integer Number

```
>>> a = 13
>>> print(type(a))
<class 'int'>
```

- Floating Point Numbers

```
>>> b = 13.5
>>> print(type(b))
<class 'float'>
```

- Complex Numbers

```
>>> c = 1+5j
>>> print(type(c))
<class 'complex'>
```

- Strings

```
>>> d = 'Hello, world!'
>>> print(type(d))
<class 'str'>
```

- Bytes

```
>>> e = b'Hello, world!'
>>> print(type(e))
<class 'bytes'>
```

- Boolean

```
>>> f = (1==2)
>>> print(f)
False
>>> print(type(f))
<class 'bool'>
```

What can we do with these data types?

- With numbers, we can use arithmetic operators:

```
+ - * / % // **
```

E.g.

```
>>> 4 + 6//2
7
```

- We can use comparison operators:

```
== != < > <= >=
```

which result in a boolean, e.g.

```
>>> 4==1
False
>>> 3<5
True
```

- With booleans, we can do logic operators:

```
and or not
```

E.g.

```
>>> True and False
False
>>> (1==2) or (2==2)
True
```

- With strings, you can concatenate

```
>>> "Hello" + ", World"
"Hello, World"
```

and you can use comparisons:

```
== != < > <= >=
```

The `print()` function

- Print is a function that prints its arguments to the terminal.
- Examples:

```
>>> print(100)
100
>>> print("Hello")
Hello
>>> a = 42
>>> print("Hello", "world!", a)
Hello world! 42
```

- You can print several things at once, by listing them in the parentheses to the `print` function, separated by commas.
- These 'several things' are called the arguments of the function.

The `if` statement

The following code gives a "Good morning!" greeting if `hour` is less than 12.

```
>>> hour=11
>>> if hour < 12: print("Good morning!")
Good morning
>>> hour=16
>>> if hour < 12: print("Good morning!")
>>>
```

It does so for two case: `hour=11` and `hour=16`.

- `if` followed by a condition and a colon will execute what is after the colon if the condition is met.

Multi-line input

If we have more than one statement to do when the condition is met:

```
>>> hour=7
>>> if hour < 12:
...     print("Good morning!")
...     print("Would you like some coffee?")
...
Good morning
Would you like some coffee?
>>>
```

- How did we enter a multiline piece of Python code at the >>> prompt?
- Well, Python noticed that the statement `if hour < 12:` requires more code, so it enters *multi-line mode*.
- Multi-line mode is indicated by `...`, and you can now continue typing code.

- To indicate that your multi-line input has been completed and should be interpreted and executed, you press Enter on an empty line.

Code blocks

If we have more than one statement to do when the condition is met:

```
>>> hour=7
>>> if hour < 12:
...     print("Good morning!")
...     print("Would you like some coffee?")
...
>>>
Good morning!
Would you like some coffee?
>>>
```

How does Python know which lines are part of the statements that are executed upon the condition being true?

- The `if` statement takes the next

  code block.

- The start of a code block is indicated by an **indented line**.
- That code block includes all lines that are indented in the same way.

User input with the `input()` function

- With the `input` function, we can ask the user to type in a value and store it in a variable.

```
>>> s=input()
...
>>> print(s)
...
```

- You can pass a string to it, which becomes the prompt from that input:

```
>>> s=input("Give a number: ")
Give a number: ...
>>> print(s)
...
```

- Regardless of the inputted value, the type of value that `input()` returns is always a string.

  You'd have to convert it yourself to a number of that's what you'd expect, e.g.

```
>>> s_as_int = int(s)
>>> s_as_float = float(s)
```

Scripting

- Okay, so we typed in the `input()` command, then we typed in a value (say `5`), and then that value was stored in `s` as a string.
- Why did we not just store the value in `s` (`s='5'`), then?
- The idea is that the input could be given by some other user, but since they'd have to be sitting right next to us as we are coding, they could type the `s='5'` statement.

```
>>> s=input("Give a number: ")
Give a number: ...
>>> print(s)
...
```

- We have arrived at a point where the interactive session has lost its utility.
- We want to create something that will execute the Python commands elsewhere without typing them in interactively.
- An app, if you wish.

Script=program=application=app

- As far as Python is concerned, they are all the same.
- It's something you can run and which performs a function.
- With running, we mean here typing `python SCRIPTNAME` on the terminal command line (i.e., after the $ sign), with `SCRIPTNAME` replaced by the name of your script.

Creating Python scripts

- Creating a Python script is as simple as storing the commands into a text file.
- Choose your editor, ensure it can save as 'plain' ASCII text. No .doc or .rtf, please.

  (E.g., nano, emacs, vi, vim, sublime text, gedit, notepad, ...)
- Make sure you understand where your editor saves your files.
  - Either save your file in the directory where your terminal is.
  - Or change directory in the terminal to the directory where you editor saves your files.
- Creating, editing, saving a text file differs per system, and is something you will have to be able to do, so let's get this working.

Hands-on #2

- Create a first script in a file called ‘first.py’.
- In your editor, type:

```
s=input("Give me a string: ")
print(s)
```

- Save it as ‘first.py’.
- Open the terminal (if not open yet).
- Go to the directory where you saved ‘first.py’ (if not yet there).
- Type, after the prompt, `python first.py`, i.e.,

```
$ python first.py
```

Make sure it works, i.e., it asks for a string, allows you to type one, and then prints it back.

SciNet
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

More about code blocks in Python

- Some statements take the next **code block**

```
for ...:                    while ...:
with ...:                   def ...:
try:                        except:
class ...:                  if ...:
elif:                       else:
```

  These statements all end in a colon (`:`).

- The start of a code block is indicated by an **indented line**.
- That code block includes all lines that are indented in the same way.
- That code block ends at a line that has a lesser (really, the previous) indentation.
- Code blocks can be nested, increasing the indentation further.

The `else` statement

What if the condition is not true, and we need a different set of statements to be executed?

Use "`else:`".

```
>>> hour=7
>>> if hour < 12:
...     print("Good morning!")
...     print("Would you like some coffee?")
... else:
...     print("Good afternoon!")
...     print("No more coffee for you!")
...
Good morning!
Would you like some coffee?
>>>
```

It's not the afternoon after hour=17!

Right, so let's fix that with "`elif`"

```
>>> hour=19
>>> if hour < 12:
...     print("Good morning!")
...     print("Would you like some coffee?")
... elif hour < 17:
...     print("Good afternoon!")
...     print("No more coffee for you!")
... else:
...     print("Good night!")
...
Good night!
>>>
```

Hands-on #3

- Take the `if/elif/else` code from the previous slide and put it in the script.
- But let the script first ask the user to input the `hour`.
- Remember to convert the `str` that `input()` returns to an `int`.
- Test the script out by running it and inputting some representative values for `hour`.
- Make it distinguish night, morning, afternoon and evening.

  (Let's just say: Night: 21-24 and 0-5, morning: 6-12, afternoon: 13-17, evening: 18-20)
- Test again.

Errors

- Taking input from a user, and then validating it, is a rather big topic on its own that we do not want to touch.
- Nonetheless, your code should to some extent be prepared for thing to go wrong.
- E.g., what if `s=input()` is suppose to give a integer, the code does `int(s)`, but the input isn't integer? We get some funny error like:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '45.1'
```

- We will talk about Python's error messages later, but the users of your script do not wish to decipher that.

Error handling

- You could check if the string is in fact a number, as there's a function for that.
- It would look like this:

```
s=input("Give me an integer: ")
if s.isnumeric():
   i=int(s)
   print(i)
else:
   print("That is not an integer!")
```

- Good, but there may be other things that go wrong in the input that we did not catch.
- An alternative is the 'try first, deal with failure later' model: exceptions.
- This take the following form

```
s=input("Give me an integer: ")
try:
   i=int(s)
   print(i)
except:
   print("That is not an integer!")
```

- You can be more specific in the except on what kind of error you're catching, but let's not worry about that now.

Hands-on #4

Create a script that:

- Read two strings, call them `astr` and `bstr`
- Check if a number, if not error, else print the sum of `astr` and `bstr`.

SciNet
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

Python's error messages

Because we cannot foresee every possible error, let's look at a typical uncaught Python error.

```
>>> print 17
  File "<stdin>", line 1
    print 17
          ^
SyntaxError: Missing parentheses in call to 'print'
```

Read the lines in the error messages carefully:

1. Something's up in line 1 in a file "<stdin>", i.e., the prompt.
2. The statement with the issue is printed, here, it is `print 17`.
3. The ^ more precisely pinpoints where there's an issue
4. The last line is most informative: there should have been parentheses in the call to 'print'.
   The type of this error is a 'SyntaxError'.

Python's error messages

Let's look at another error message:

```
>>> print(seventeen)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'seventeen' is not defined
```

Read the lines in the error messages carefully:

1) *What's a traceback?*

   When the error occurs in the execution step, several function may be called before the error, and the traceback would show these.

2) Here, the error occurs in line 1 in a file "`<stdin>`", i.e., the prompt, but before a function has been called, i.e., on the "`<module>`" level.

3) Again, the last line is most informative: the variable seventeen has not been defined (yet?).
   The type of this error is a 'NameError'.

Python's error messages

Here's another one:

```
>>> a = 11
>>> b = '17'
>>> c = a + b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

What was going on here?

Section 4

Repetitions/Loops

Try again

- In the read-an-int example, it would be nice to start over if the user didn't enter an integer.
- A 'go to beginning' statement does not exist in Python (no 'go-to's in fact), but loops are.
- Loops are repetitions of a code block for different, given cases, or until a condition is fulfilled.
- So we could 'loop' (do the same thing over and over again) until the entered string is an integer.
- This would be a `while` loop.

  (The other kind of loop is a `for`, which we will see later)

While loop

- In the read-an-int example, it would be nice to start over if the user didn't enter an integer.
- We could 'loop' until the entered string is an integer.

This is one way:

```
haveint=False
while not haveint:
  s=input("Give me an integer: ")
  try:
   i=int(s)
   haveint=True
  except:
   print("That is not an integer, try again!")
print(i)
```

- At the start of the while loop, `haveint` is checked, and Python enters the the code block that belongs to `while` (the "body of the loop")
- If `i=int(s)` succeeds, `haveint` is set to True.
- `haveint` is checked at the next iteration.
- Note that `print(i)` is outside the loop body.

Escaping the loop

- If the expression after `while` is not true after the loop body is executed, the loop stops.
- The loop can also be stopped at any time in the loop body with the **`break`** keyword.

In both cases, execution of the script continues with the next non-indented line of code.

So instead of:

```
haveint=False
while not haveint:
  s=input("Give me an integer: ")
  try:
   i=int(s)
   haveint=True
  except:
   print("That is not an integer, try again!")
print(i)
```

We could also have used:

```
while True:
  s=input("Give me an integer: ")
  try:
   i=int(s)
   break
  except:
   print("That is not an integer, try again!")
print(i)
```

Note: break stops the loop, but not the script. The `exit()` function can stop a script.

Hands-on #5

Create a script that:

- Reads two strings, call them `astr` and `bstr`
- Checks if they are numbers, if not, let user know which one is wrong, and let them enter both numbers again.
- If they both contain numbers, print the sum of integer values of `astr` and `bstr` and exit.

Section 5

Functions

Functions

- The solution to the last hands-on problem likely contain repeated code to check whether `astr` and `bstr` are numbers.
- Repeated code is bad: can make mistakes in twice as many lines of code.
- **Functions** are bits of reusable code.
  (Not the same as mathematical functions, mind you!)
- They are created with the `def` keyword.

Silly example

```
# example_silly.py
def printfruit():
    print("apples and oranges")
printfruit()
printfruit()
```

```
$ python example_silly.py
apples and oranges
apples and oranges
```

Function arguments

- If functions did the same thing every time (like our example), they'd be of little use.
- The `print` function is an example of a less-trivial function.
- Depending on its *arguments*, the print function does something else.
- When we write our own functions, we can allow for one or more arguments too.
- The function definition must specify the names of the argument.

```
# example_silly_argument.py
def printfruit(s)
    print (s, "apples and oranges")
printfruit("I like")
printfruit("I do not like")
```

```
$ python example_silly_argument.py
I like apples and oranges
I do not like apples and oranges
```

- `printfruit` takes one argument called `s`.
- Inside the function, `s` acts like a variable.
- After the function definition, we can use `printfruit` with different arguments.

More function arguments

- We can also have multiple function arguments.
- Simply supply multiple names for function arguments, separated by commas.

```
#example_fun_moreargs.py
def comparable(a,b):
    if a != b :
       print ('you cannot compare ', a, 'and', b)
    else:
       print (a, 'and', b, 'are comparable.')
comparable('apples','apples')
comparable('apples','oranges')
```

```
$ python example_fun_moreargs.py
apples and apples are comparable
you cannot compare apples and oranges
```

SciNet
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

Function output

- So far our functions took arguments, i.e., input, but they produced output on the terminal.
- Functions are more useful if they produce output that can be put in a variable.
- To have your function produce such output, use the `return` statement.
- Whatever follows the return statement of a function becomes the function's return value. The function exits at the return statement.
- To use that return value to a variable, use the function call as if it's a variable.

Silly example

```
# example_fun_output.py
def addone(x):
    return x+1
a=10
b=addone(a)
print(a,b)
```

```
$ python example_fun_output.py
10 11
$
```

Hands-on #6

- Write a function `solve` that returns the solution of the equation

$$y = ax + b$$

given real numbers a, b, and y (in that order).

- Hint: The mathematical solution is $x = (y - b)/a$.
- The function should be in a Python script, and at the end of the script, the following test cases should be performed:

`print(solve(1, 3, 4))` should give `1.0`.

`print(solve(0.2, 0.3, 0.3))` should give `0.0`.

`print(solve(0.1, 0.3, -0.8))` should give `-11.0`.

More on functions

There's a lot more to be said about functions, the scope of variables, default values, variable number of arguments, keyword arguments, ...

But we need to go over some other stuff first.

Section 6

Composite data types

Lists

- A list is a collection of objects.
- We have not talked about objects before, but any data of all the types we have introduced count as objects: integers, strings, floats.
- We create a list by putting objects between square parentheses `[]`, separated by commas, e.g.,

```
>>> lst = [10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 'blast off!']
```

- List elements do not all have to be the same type.
- Lists are objects, so list elements can be lists themselves.

What can we do with these lists?

- We can access elements using the notation `LISTNAME[INDEX]`.

  E.g.:

```
>>> lst = [10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 'blast off!']
>>> print(lst[0])
10
>>> print(lst[10])
blast off!
```

  Note that the first element has index 0.

  (You can think of the index as an offset from the beginning of the list.)

- We can reassign elements of the list, too:

```
>>> lst[10] = 'abort'
>>> print(lst)
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 'abort']
```

What can we do with these lists?

- You can add an element to the end with `append` method:

```
>>> lst.append('not ready')
>>> print(lst)
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 'abort', 'not ready']
```

- You can remove an element by index with the `pop` method:

```
>>> lst.pop(2)
8
>>> print(lst)
[10, 9, 7, 6, 5, 4, 3, 2, 1, 'abort', 'not ready']
```

Note that the removed element is returned by pop.
(`del lst[2]` would also have worked, but would not have returned removed element).

- The current length of the list is obtained from the `len` function.

```
>>> print(len(lst))
11
```

What are these "Methods"?

Regular functions take arguments in parentheses and return something:

```
>>> x = f(y,z)
```

Methods are functions whose first argument is placed in front of the function name:

```
>>> x = y.f(z)
```

- Of course, it was just a syntax variation, that would be rather silly.
- What "`y.f(z)`" indicates is that the method `f` is part of `y`.
- So `y` must be an object that has `f` as a method.
- Different object types can have their own `f` method that is appropriate for that data type.
- Data and functions acting on that data are thus combined; that is one of the characteristics of "object oriented programming".
- One can find the methods of a variable `x` with "`dir(x)`".

Hands-on #7

- Create a list with one element, '7'.
- Add the following elements:
  - `17.5`
  - `-1`
  - Hello, world.
- Remove the third element.
- Print the list

Loops with lists

- It is rather common to have to go over a list and do something with every element.
- This is a repetition, i.e., a loop.
- We could write

```
i = 0
while i<len(lst):
   x=lst[i]
   do_something_with(x)
   i=i+1
```

- But Python can do that with a `for` loop:

```
for x in lst:
   do_something_with(x)
```

Hands-on #8

Write a script that:

- Let the user input a string.
  If it is a number, put it in a list called `lst`.
- Let the user input further strings.
  If it is a number, append it to the list `lst`.
  If it is the string '`exit`', stop asking for user input.
- After each number that is entered, print the average of the numbers so far.
- Before stopping, print all the numbers and their average once more.

Other handy `list` manipulations

- `lst.index(value)`: Find the index of `value` in `lst`.
- `lst.count(value)`: Count number of occurrences of `value` in `lst`.
- `lst.extend(otherlist)`: Append all elements of otherlst.
- `lst.insert(index,object)`: Insert object in list at position `index`.
- `lst.remove(value)`: Remove the first element that is equal to value in `lst`.
- `lst.copy()`: Create a copy of `lst`.
- `lst.clear()`: Remove all elements from `lst`.
- `lst.reverse()`: Reverse the `lst`.
- `element in list`: Check if `element` is in `lst`.
- `lst.sort()`: Put `lst` in sorted order.
- `sorted(lst)`: Create a sorted version of `lst`.

List-like data types

- **Sets:** are lists in which an element can only occur once.

  Elements in a set an unordered (so no indexing).

  Sets are defined with curly braces:

```
>>> s = {5, 4, 6, 5}
>>> print(s)
{4, 5, 6}
```

- **Tuples:** are lists that cannot be changed.

  Tuples are denoted with parentheses.

```
>>> t = (1,2,1)
>>> print(t)
(1, 2, 1)
```

- **Generators:** are lists generate their next element upon request.

```
>>> r = range(4)
>>> print(r)
range(0, 4)
>>> for x in r:
...   print(x)
...
0
1
2
3
```

- **Dictionaries:** are key-value hash tables.

Dictionaries

Key? Value? Hash? Table?

- In Python, a dictionaries (`dict` for short) is a look-up table.
- Think of the directory of a building:
- This would need to allow look-up for 'Airbnb', 'SciNet', ...
- And give the suite number when selected.

In Python:

```
>>> directory={'Airbnb': 10, 'SciNet': 1140}
>>> print(directory['SciNet'])
1140
```

Dictionaries

```
>>> directory={'Airbnb': 10, 'SciNet': 1140}
>>> print(directory['SciNet'])
1140
```

- A `dict` translates a 'key' to its associated 'value'.
- In the above example, the keys are 'Airbnb' and 'SciNet', and the values are 10 and 1140.
- You can give the key in square brackets to get the value out (a bit like a list).
- Keys must be unique, but can be integers, strings, ...
- Values can be anything.
- Other names for these structures:

  Associate Array, Map, Hash Map, Unordered Map

What can we do with these dicts?

- Look up values:

```
>>> directory['SciNet']
1140
```

- Get all the keys:

```
>>> directory.keys()
dict_keys(['SciNet', 'Airbnb'])
```

- Get all the values:

```
>>> directory.values()
dict_values([1140, 10])
```

- Add key-value pair:

```
>>> directory['TTC'] = 0
```

- Loop over all keys

```
>>> for k in directory:
...   print(k)
...
SciNet
Airbnb
TTC
```

- Loop over key-value pairs

```
>>> for k,v in directory.items():
...    print (k,v)
...
SciNet 1140
Airbnb 10
TTC 0
```

Hands-on #9

- This is the price list of some groceries:

  potatoes: $3 / lbs
  squash: $5 / lbs
  onion: $1 / lbs
  corn: $2 / lbs
  cabbage: $5 / lbs

- You have got a shopping list:

  2 lbs potatoes
  1 lbs onion
  0.5 lbs cabbage

- Write a script that stores the price list and the shopping list in `dicts`.
- Make it compute the total amount to pay the cashier.

List comprehension

- List comprehensions are a short hand way of creating lists.
- They combine a for loop, appends, and if statements.
- Example: list of all squares that are divisible by 4 and are less than 100.

without list comprehensions:

```
squarelist=[]
for i in range(100):
    i2 = i**2
    if i2%4 == 0:
       squarelist.append(i2)
```

with list comprehensions:

```
squarelist=[i**2 for i in range(100) if i**2 % 4 == 0]
```

General syntax:

```
[ <EXPRESSION> for <VARIABLE> in <LIST-LIKE> if <CONDITION>]
```

The `if` is optional, e.g. `[i**2 for i in range(4)]` gives `[0,1,4,9]`.

Section 7

Documentation and comments

Code is for humans

- So far, we have been writing **code for the computer.**
  (We tried to make the computer do something.)
- **Programs are meant to be read by humans and only incidentally for computers to execute.**
  (Harold Abelson, *Structure and Interpretation of Computer Programs*)
- How so?
  - Programmers spend more time reading someone else's code than writing their own.
  - There's no such this as a one-off script.
    (If you have saved a script as a file, it's no longer one-off, and you or someone else will eventually use it again and want to adapt it.)
- Readability, documentation, and maintainability very important.

SciNet
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

How do you code for humans?

1. Write clear code.
2. Comment your code.
3. Document your code.

1. Write clear code: general principles

- **KISS: Keep It Simple, Stupid!**
  - Do not write code that is more complicated than necessary.
  - It will take too long for the next programmer of your future self to decode.
  - Your cleverest code will need someone cleverer than you to debug.
- **DRY: Don't repeat yourself.**

  Use functions to extract repeated code.
  $\Rightarrow$ Less code to figure out, less possible bugs, less code to maintain.
- **Separation of concerns**

  Each function should do only one thing, and do it well.

  This makes it easier to figure out, bugs become less complex, documentation becomes easier to write, and code can be reused in more situations (DRY).

1. Write clear code: style matters

A couple of code style guidelines can help too:

- Clear variable and function names
  `b` → `calls_per_postal_code`
- One statement per line
- Use a consistent style
- Prefer small functions over long ones (as long as they perform a non-trivial task).
- Don't reinvent the wheel; use existing functions and packages.
- No cleverness.
- Use comments and documentation

2. Comment your code

- Comments start with #; anything after that is a comment.

```
amount_flour = 2.2 # amount in pounds
brand = 'Brand #1' # string contains '#'
```

- Keep comments succinct.
- Describe ***why*** your the code is doing something.

```
a_is_prime = all(a%i for i in range(2,a))
  # brute force determination of whether
  # a is prime (was easier to code than
  # a more sophisticated algorithm).
```

- Describe on a high level what parts of the code are doing.

```
# Get to user input an integer
while True:
  try:
    input_string = input("Enter an integer a=")
    a = int(input_string)
    break
  except:
    print("Not an integer; try again")
# Determine if integer a is prime
a_is_prime = all(a%i for i in range(2,a))
  # brute force determination of whether
  # a is prime (was easier to code than
  # a more sophisticated algorithm).
# Report back result to screen
print("a is prime?", a_is_prime)
```

SciNet
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

3. Document your code

- You *would* also add comments describing what a function does, what parameters they take, and what they return.
- This would be the first line of defence in documenting that function.
- However, Python has a separate mechanism for such function documentation, called a docstring.
- Doc-strings are placed at the first line of the function.
- The docstring is returned by the `help` function.

```
>>> help(find_second)
```

```
>>> def find_second(searchin, forthis):
...   """Finds the 2nd occurance of a string.
...
...   Args:
...     searchin (str): string to search in.
...     forthis  (str): string to search for.
...
...   Returns:
...     int: The index in the search where the
...          second occurance starts.
...          -1 if there is no 2nd occurance.
...   """
...   # code would follow
...
```

The triple quotes are the Python syntax for multi-line strings.

How to code for humans, continued

1. Write clear code
2. Comment your code
3. Document your code

Some schools of thought say that #2 and #3 are unnecessary if you write "self-documenting code".

Humbug!

Points #2 and #3 can be done simultaneously with Python **only up to a point**.

For more complex code, you will need to write files like `README`, `doc.txt`, `manual.pdf` ...

Section 8

Intermezzo: Shebangs and PATH

Shebang comment

- Under Linux and MacOS, one often sees Python scripts that start with a "shebang" line:

```
#!/usr/bin/python
```

or (much better)

```
#!/usr/bin/env python3
```

- Such a first line tells the operating system that this is a Python script.
- Being identified as a Python script, it can be executed from the terminal command line:

```
$ ./mypythonscript.py
```

instead of

```
$ python mypythonscript.py
```

- For this to work, you might need to tell the OS additionally that this file may be executed:

```
$ chmod +x mypythonscript.py
```

Intermezzo: PATH

- On Linux or Mac, PATH is an environment variable containing a list of directories, separated by colons (":").

```
$ echo $PATH
/x/opt/base/python/3.11.5/bin:/x/core/bin:
/usr/local/bin:/usr/bin:/bin:/home/rzon/bin
```

- "echo" is how you print on the command line
- "$" + a name is how you access a variable

- The PATH specifies the directories where the system should look for executable files when you run a command on the command line.
- You can modify the PATH variable temporarily in the current shell session:

```
$ PATH="$PATH:."
$ mypythonscript.py
```

- "." stands for current directory
- There are no spaces around '='
- Keeps the previous path using $PATH

- To make this permanent, you must add this to a shell configuration files, like ~/.bashrc or ~/.zshrc.

Section 9

Modules

What are modules and packages?

- ***Modules*** are Python files.
- Modules are meant to be imported into other Python files or scripts.
- Convenient way to store functions that you might use in multiple projects.
- ***Packages*** are modules that are packaged for distribution and installation.

Example

Module file:

```
# file: yearprop.py
"""Deal with properties of calendar years."""
def is_leap_year(year):
  """Determines if a give year is a leap year.
  Argument year is the year to investigate.
  Returns True is year is a leap year, else
  False.
  """
  ...
```

Usage in another script:

```
# file: yearquery.py
"""Ask for years and say if they're leap years"""
import yearprop

# use the function yearprop.is_leap_year
year = int(input("Give a year"))
if yearprop.is_leap_year(year):
  print(year,"is a leap year")
else:
  print(year,"is not a leap year")
```

Module example expanded

Module file:

```
# file: yearprop.py

"""Deal with properties of calendar years."""

def is_leap_year(year):
  """Determines if a give year is a leap year.
  Argument year is the year to investigate.
  Returns True is year is a leap year, else
  False.
   """
  if year%400 == 0:
    return True
  elif year%100 == 0:
    return False
  elif year%4 ==0:
    return True
  else:
    return False
```

Usage in another script:

```
# file: yearquery.py
"""Ask for years and say if they're leap years"""

import yearprop

while True: # keep processing new input
  # keep asking input until we get an integer
  while True:
    try:
      year = int(input("Year (0 to stop)? "))
      break
    except:
      print("That is not an integer.")
  # process input
  if yearprop.is_leap_year(year):
    print(year, "is a leap year!")
  else:
    print(year, "is not a leap year.")
  if year==0: break # stop processing input
print("Done")
```

Modules - Details

- We do not have to specify `.py` in the `import` statement.
- The file name without `.py` is the name of the module.
- The module file has to be in the same directory as the script, unless you install it (*later*)
- When putting functions in a module and importing that module, it gets put in the **namespace** of the modules. The name of the namespace is the name of the module.

  (in the example, the namespace was `yearprop`)

You can change the namespace that the modules functions end up in

- Changing the name of the module: `import yearprop as yp`
- Importing specific functions: `from yearprop import is_leap_year`
- Importing everything: `from yearprop *`

Hands-on #10

- Type out the codes from slide 81, store them in files `yearprop.py` and `yearquery.py`
- Make sure they work on the terminal command line, i.e.

```
$ python yearquery.py
Year (0 to stop)? 1972
1972 is a leap year!
Year (0 to stop)? 2020
2020 is a leap year!
Year (0 to stop)? 2023
2023 is not a leap year.
Year (0 to stop)? 0
Done
```

- We will use this code later again.

Standard Packages

- There are many, many, many standard packages.
- We will only get to look at a few very basic ones:
  - `sys`: system specific parameters and functions (command line arguments, exit, path, ...)
  - `os`: operating system stuff (chdir, stat, getenv, listdir, walk, ...)
  - `shutils`: High level file operations (copyfile, copytree, rmtree, move, ...)
- For more standard packages, see https://docs.python.org/3/library/index.html

- In addition, there are non-standard packages in the pypi repository (https://pypi.org), that you can install with the `pip` command on the terminal command line (not within Python).
- These modules are put in a special location that Python knows about, so they do not have to reside in the directory of your script.
- You can control those locations by using *Virtual Environments*.

Section 10

Virtual Environments

Virtual environments solve a number of problems

- Many python installations are owned by the system, so you cannot install packages
- Or if you try, it will only work for you.
- And you cannot have different several projects with different package requirements.
- Because many python packages conflict with one another, and you cannot have different versions of the same package installed at the same time.

SciNet
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

Virtual environments solve a number of problems

How?

- Create a virtual environment directory with a local python "installation" based off of your main python installation.
- When the environment is activated, PATH is set so that `python` and `pip` is used and packages get installed in that directory.
- When you deactivate the environment, you have the old python back.
- And then you can create and activate more environments.

SciNet
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

How, exactly?

Create an environment

```
$ virtualenv DIRNAME
```

Activate an environment

```
$ source DIRNAME/bin/activate
```

Install packages

```
(DIRNAME)$ pip install PACKAGENAME
```

Deactivate the environment

```
$ deactivate
```

Conda?

Conda is similar, but a bit powerful yet more invasive (i.e., some other things may not work in your shell).

Hands-on #11

1. Create a virtual environment called "bio".
2. Activate it.
3. Install the biopython package.
4. Start `python`, then `import Bio`, then `help(Bio)`.
5. Deactivate the environment.
6. Repeat step 4, which should fail.
7. Reactivate the environment.
8. Repeat step 4, which should succeed.

SciNet
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

Section 11

Other Methods for Input in Python

Input in Python

- The `input` function is what we have used for user input so far.

  The user needed to type something in.

  That's not very automated!

Let's look at two more common ways of providing input to a Python script that lend themselves better to automation:

1. Command line arguments
2. Files

Section 12

Command-line arguments

What are Command-line arguments?

The terminal command line execution of Python scripts, e.g.

```
$ python yearquery.py
```

can be augmented with arguments on the command line

```
$ python yearquery.py 1972 2000 2017
```

But this will not 'just work'. The script will need to be expecting command line arguments and deal with them.

Command-line arguments: `sys.argv`

- A script needs to expect command line arguments and deal with them, or those arguments will be ignored.
- The standard `sys` module contains a variable called `argv` which is a list of all command line arguments, called `argv`.
- To use it, import the `sys` module and use `sys.argv` as a list.

E.g.

```
# file: sys_argv_example.py
import sys
print("You gave",len(sys.argv),"arguments")
for arg in sys.argv:
    print(arg)
```

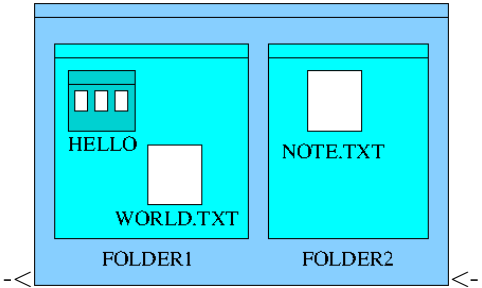
Hands-on #12

- Take the Python codes from hands-on #10
- Rewrite `yearquery.py` to use, instead of the input from `input()`, the `sys.argv` list.
- The module file `yearprop.py` should not have to be changed for this.

Section 13

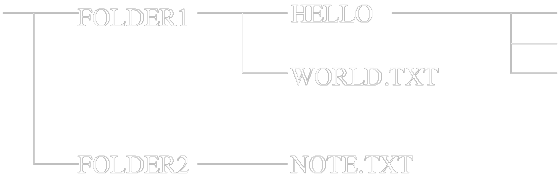
File system and I/O

File System: Concepts



- Files contain your data
- Files organized in directories/folders
- A directory is a file too
- Path: sequence of folders to get to a file

Tree:



Files:

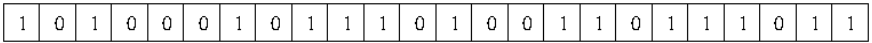
```
FOLDER1/WORLD.TXT
FOLDER2/NOTE.TXT
FOLDER1/HELLO/...
```

Computer Data Storage

Media:

- Memory
- Disks
- Flash (USB)
- DVD
- Tape
- ...

All media are essentially linear sequence of bits:



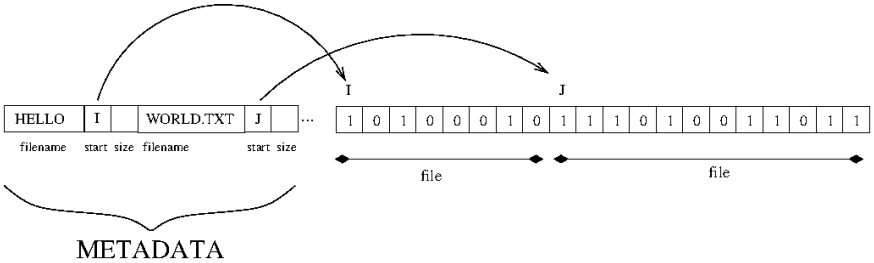
In and of itself, this is useless. What do these bits mean?

File systems

- Many non-volatile media use a file system
- A file system is a way to give meaning to the sequence of bytes.
- This entails storing data describing the meaning of the data: **metadata**

Files

- Storage media is often subdivided into files
- Files have a name, a size and possibly other metadata
- Let's say that the metadata for the files is stored at the beginning of the storage media, e.g.



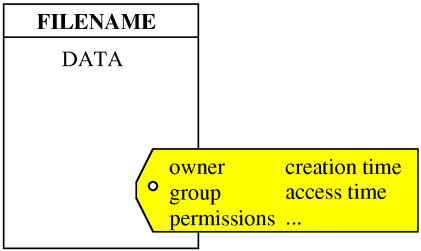
Metadata

Describes file properties:

- File name
- Within the file system: location on disk, size, etc.
- File type

  (extensions/magic identifiers)
- Owner, group
- Creation, access and modification times
- Read/write permissions

  (user, group, world, other access control)



Directories or Folders

So we have files now, but this can get unorganized quickly.
Imagine looking for the file 'NOTE.TXT' in a list of 10,000,000 files.

Directories

- Like special files that contain a list of (metadata for) other files.
- A directory can contain other directories, leading to a tree.

I/O Operations

What really happens if we open a file, write to it, etc.?

Opening a file:

1. Find the file in the directory

   Or create a new entry in the directory
2. Check permissions on the file
3. Find the location of the file on disk
4. Initialize a file ‘handle’ and file ‘pointer’

I/O Operations

What really happens if we open a file, write to it, etc.?

Writing to a file:

1. Convert data to a stream of bytes.
2. Put those bytes in a buffer.
3. Update file pointer.
4. If buffer full: write to file

I/O Operations

What really happens if we open a file, write to it, etc.?

Reading from a file:

1. If data not in buffer: read data into a buffer
2. Read bytes from buffer into variable, performing any needed conversion.
3. Update file pointer.

I/O Operations

What really happens if we open a file, write to it, etc.?

Closing a file:

1. Ensure buffers are flushed to disk
2. Update any metadata.
3. Release buffers associated with the file handle.

The Goal of the Game is Minimizing IOPS

- Disk I/O is usually the slowest part of a pipe line.
- If manipulating data from files is most of what you do, try and minimize IOPS.

Bad

Writing out a byte-by-byte, reopening the file each time

```
s = 'Hi world\n'
for c in s:
  f = open('hiworld.txt','a')
  f.write(c)
  f.close()
```

Good

Writing out a string in one fell swoop.

```
s = 'Hi world\n'

f = open('hiworld.txt','w')
f.write(s)
f.close()
```

- **Work in memory and reuse data if you can.**

What's in a file?

Text:

- Seems attractive: you can just read it.
- Must assign a bit pattern to each letter or symbol.
- For numerical data, representation in base 10 must be computed.

Binary:

- Usually: use same byte-representation on disk as the computer.
- Can suffer from portability.
- Some binary formats include info on the data, e.g.: hdf5 and NetCDF.

Encoded:

- Various non-native, binary looking formats, e.g. pickle.
- Might be used to store non-trivial data structures.
- Example: Python's pickle (later).

SciNet
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

Text format

- ASCII Encoding: 7 bits = character
- 128 possible, but only 95 printable characters
- Uses 8-bit bytes: storage efficiency 82% at best.
- ASCII representation of floating point numbers:
  - Needs about 18 bytes vs 8 bytes in binary: **inefficient**
  - Representation must be computed: **slow**
  - **Non-exact** representation

ASCII

integers	characters
32	(space)
33-47	`!"#$%&'()*+,-./`
48-57	`0-9`
58-64	`:;<=>?@`
65-90	`A-Z`
91-96	`[\]^_`
97-122	`a-z`
123-126	`{|}~`

Text Encodings

ASCII: 7 bit encoding. For English.
Latin-1: 8 bit encoding. For western European Languages mostly.
UTF-8: Variable-width encoding that can represent every character in the Unicode character set.
Unicode: standard containing more than 110,000 characters.

You can tell Python what encoding your scripts are in as follows:

```
# -*- coding: utf-8 -*-
s = "Comment ça va?"
print(s)
```

Though, in fact , the default encoding in Python 3 is utf-8.

Binary Output

- Output the numbers as they are stored in memory
- Why bother? Fast and space-efficient.

Writing 128M doubles:

ASCII	173 s
binary	6 s

- Not human readable.

But is that really so bad? If you have 100 million numbers in a file, are you going to read them all?

Some best practices concerning I/O

- If your data is not text, do not save it as text.
- Choose a binary format that is portable.
- Minimize IOPS: write/read big chunks at a time, don't `seek` more than needed, try to reuse data or load more in memory.
- Don't create millions of files: unworkable and slows down directories.
- Stick to letters, numbers, underscores and periods in file names.

SciNet
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

Python modules/packages for files

- built-in Python file objects
- `os`, `os.path`
- `shutil`
- `pickle`, `shelve`, `json`
- `zipfile`, `tarfile`, ...
- `csv`, `numpy`, `scipy.io.netcdf`, `pytables`, ...

Section 14

Directories and Files Basics

SciNet
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

Directories

List:

```
>>> import os
>>> os.listdir(".")
['FOLDER1', 'abc.txt', 'FOLDER2']
```

```
>>> [f for f in os.listdir() if os.path.isdir(f)]
['FOLDER1','FOLDER2']
```

Create:

```
>>> os.mkdir('FOLDER')
```

Change current directory:

```
>>> os.chdir('FOLDER')
>>> os.chdir('..')
```

Where am I?

```
>>> os.chdir('FOLDER')
>>> print(os.getcwd())
C:\Users\rzon\FOLDER
```

Backslash or forward slash?

- Linux and Mac prefer the (forward) slash / to separate directories.
- MS Windows prefers backslash \ for the same purpose. It also separates file trees by file volume (`C:`, `D:`, ...).

What to do if you want to write cross-platform code?

1. MS Windows will accept the forward slash as well, except on the command-line, so you could use that in Python code.
2. You can also use `os.sep` which is set to the operating system's preferred choice.
3. You can assemble and disassemble paths using `os.path.join` and `os.path.split`.

Write to a text file

Writing

```
>>> import os
>>> f = open(os.path.join('FOLDER','WORLD.TXT'),'w')
>>> s = "Hello"
>>> print(s,file=f)
>>> f.close()
```

Appending

```
>>> import os
>>> f = open(os.path.join('FOLDER','WORLD.TXT'),'a')
>>> s = "world"
>>> print(s,file=f)
>>> f.close()
```

Read a text file

Read the whole file

```
>>> import os
>>> f = open(os.path.join('FOLDER','WORLD.TXT'),'r')
>>> s = f.read()
>>> print(s)
Hello
world
>>> f.close()
```

Read the whole file by lines

```
>>> import os
>>> f = open(os.path.join('FOLDER','WORLD.TXT'),'r')
>>> s = f.readlines()
>>> print(s)
['Hello','world']
>>> f.close()
```

Read a text file

Read one line at a time

```
>>> import os
>>> f = open(os.path.join('FOLDER','WORLD.TXT'),'r')
>>> s = f.readline()
>>> print(s)
Hello
>>> s = f.readline()
>>> print(s)
world
>>> f.close()
```

Read/Write

```
>>> import os
>>> f = open(os.path.join('FOLDER','WORLD.TXT'),'r+')
>>> f.seek(1)
>>> print('i ', file=f)
>>> f.seek(0)
>>> s = f.readline()
>>> print(s)
Hi lo
>>> f.close()
```

Hands-on #13

- Write a Python script that write the numbers 1972, 2000, and 2017 on separate lines to a file called `years.dat`.
- Take the `yearprop.py` from hands-on #10 (see course website).
- Create a script `yearqueryfromfile.py` to read in the years from the `years.dat` file and report if they are leap years using the (unmodified) yearprop module.

Glob

The glob package does only one thing: it finds all files or paths matching a specific Unix-style regular expression pattern, and returns them in a list.

```
>>> import glob
>>> f = glob.glob('*/*.TXT')
>>> print(f)
['FOLDER\\NOTE.TXT', 'FOLDER\\WORLD.TXT']
>>>
```

os.path

There are a number of useful file and directory-testing functions in os.path.

```
>>> print(f)
['FOLDER/NOTE.TXT', 'FOLDER/WORLD.TXT']
>>> import os
>>> print(os.path.isfile(f[0]))
True
>>> print(os.path.isdir(f[1]))
False
>>> print(os.path.abspath(f[1]))
'C:\Users\rzon\FOLDER\WORLD.TXT'
>>> print(os.path.expanduser('~'))
'C:\Users\rzon'
```

If you're looking for a directory-testing function, it's likely in os.path.

Example: Copy files

Text file

```
>>> f = open("file1.txt","r")
>>> g = open("file2.txt","w")
>>> for line in f:
>>>   g.write(line)
>>> f.close()
>>> g.close()
```

Binary file

```
>>> f = open("file1.bin","rb")
>>> g = open("file2.bin","wb")
>>> chunk = f.read()
>>> g.write(chunk)
>>> f.close()
>>> g.close()
```

`shutil` file/directory management

- Do we really have to open a file read it line by line, write it, and close the file just to copy a file in Python?
- In the command shell, you'd do that with a simple `cp` or `copy` command.
- In Python, you get shell-like functionality from the `shutil` package.

```
>>> import shutil
>>> shutil.copyfile('file1.txt','file2.txt')
```

Main `shutil` functions

- `copyfile`
  Copy content of one file to another file.
- `copymode`
  Copy permissions of a file or directory to another.
- `copystat`
  Copy permissions and time-stamps of a file or directory to another.
- `copy`, `copy2`
  Copy content and permissions (and time-stamps, for `copy2`).
- `move`
  Move a file or directory to another place in the file tree.
- `copytree`
  Recursively copy a directory.
- `rmtree`
  Recursively remove a directory.

SciNet
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

Output formats: Pickle

- Base64 encoding using readable ASCII
- Portable for the same version of Python.
- In the `pickle` module.
- Flexible, can **serialize** any structure.

```
>>> import pickle,os,numpy
>>> a = numpy.zeros((1000,1000))
>>> f = open('a.pickle','wb')
>>> pickle.dump(a,f)
>>> f.close()
>>> print(os.path.getsize('a.pickle'))
32000196
>>> g = open('a.pickle','rb')
>>> b = pickle.load(g)
>>> g.close()
```

Output formats: Shelve

- You can pickle multiple variables in one file, but you must retrieve them sequentially.
- `shelve` allows to store multiple variables in one file, indexed by name, so you can retrieve just the variable you want.

```
>>> import shelve,numpy
>>> a = numpy.zeros((1000,1000))
>>> b = {'b':'bb','c':'cc'}
>>> f = shelve.open('b_and_c')
>>> f['a'] = a
>>> f['b'] = b
>>> f.close()
>>> g = shelve.open('b_and_c')
>>> readb = g['b']
>>> g.close()
>>> print(readb['b'])
'bb'
```

Output formats: CSV Format

- Comma Separated Values
- Common format for import/export
- Human readable

Sample csv file (data.csv):

```
3,4,5
4,3,2
5,6,7
```

Reading using the csv module

```
>>> import csv
>>> f = open('data.csv','r')
>>> s = csv.reader(f)
>>> a = [row for row in s]
>>> print(a)
[['3', '4', '5'], ['4', '3', '2'],
 ['5', '6', '7']]
```

. . . and the numpy module

```
>>> import numpy as np
>>> a = np.genfromtxt('data.csv',
...                   delimiter=',')
>>> print(a)
[[ 3.   4.   5.]
 [ 4.   3.   2.]
 [ 5.   6.   7.]]
```

Output formats: Json

- JSON (JavaScript Object Notation) is a lightweight data-interchange format
- Human readable

Reading

```
>>> import json
>>> f = open("data.json","r")
>>> b = json.load(f)
>>> f.close()
>>> print(b)
[[3, 4, 5], [4, 3, 2]]
```

data.json

```
[[3,4,5],
[4,3,2]]
```

Writing

```
>>> import json
>>> f = open("newdata.json","w")
>>> b = [[3, 4, 5], [4, 3, 2]]
>>> json.dump(b,f)
>>> f.close()
```

newdata.json

```
[[3.4.5], [4,3,2]]
```

Hands-on #14

Download the 'bunchofiles.zip' file from https://scinet.courses/1305.

Unzip it in a directory on your computer.

Now create Python script that:

1. Finds all files with an extension .csv in that directory.
2. The script should move those .csv files to a new subdirectory called 'csv_files'.

Section 15

Conclusion

What have we learned?

- A program is a set of instructions to tell a computer how to automate a task.
- Programming is the activity of writing such programs.
- Programming instructions in Python:
  - Variables
  - Data types: basic ones as well as lists and dicts
  - Conditionals
  - Loops
  - Functions
  - Modules
  - Keyboard input and screen output
  - Virtual environments
  - Command line arguments
  - Input and output using files
  - Error handling

SciNet
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

Programming Best Practices

Practices we covered in this minicourse

- Write code for humans
- Modular programming
- Using libraries
- Comment and document code
- Minimize I/O operations

Other good practices to get acquainted with

- Avoid global variables
- Version control (e.g. git)
- Defensive programming
- Integrated and unit testing
- Performance tuning (Python is slow compared to compiled languages).

SciNet
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

Further Learning Resources

- https://www.learnpython.org

  Goes from basic to advanced.

- https://www.w3resource.com/python-exercises

  Has hundreds of exercises to hone your Python coding skills.