STATS 701
Data Analysis using Python

Lecture 6: Files

Persistent data

So far, we only know how to write “transient” programs

- Data disappears once the program stops running

Files allow for **persistence**

- Work done by a program can be saved to disk...
- ...and picked up again later for other uses.

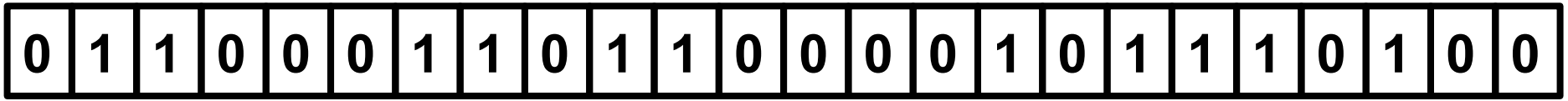
Examples of persistent programs:

- Operating systems
- Databases
- Servers

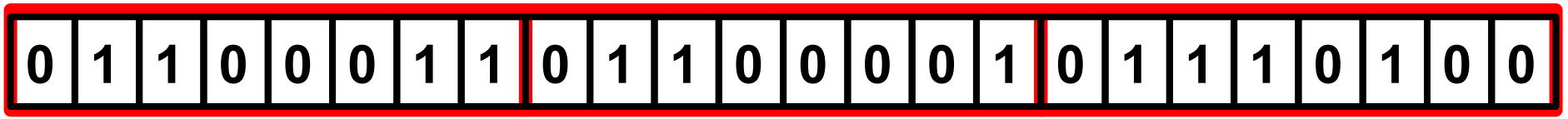
Key idea: Program information is stored permanently (e.g., on a hard drive), so that we can start and stop programs without losing **state** of the program (values of variables, where we are in execution, etc).

Reading and Writing Files

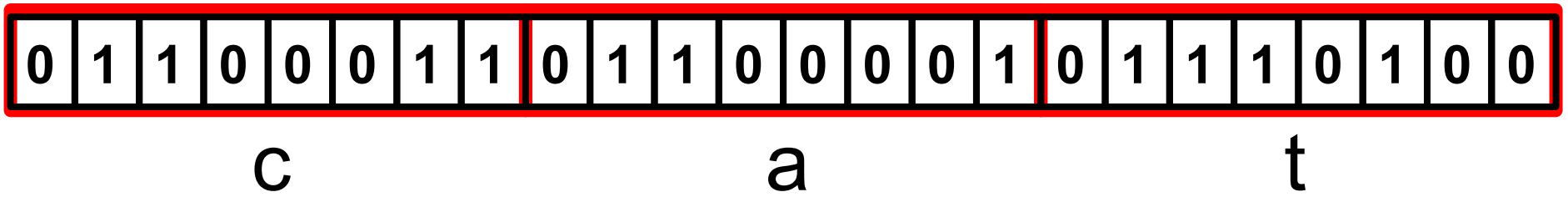
Underlyingly, every file on your computer is just a string of bits…



...which are broken up into (for example) bytes…



...groups of which correspond to (in the case of text) characters.



Reading files

This is the command line. We'll see lots more about this later, but for now, it suffices to know that the command `cat` prints the contents of a file to the screen.

```
keith@Steinhaus:~/demo$ cat demo.txt
This is a demo file.
It is a text file, containing three lines of text.
Here is the third line.
keith@Steinhaus:~/demo$
```

```
1 f = open('demo.txt')
2 type(f)
```

```
_io.TextIOWrapper
```

Open the file `demo.txt`. This creates a **file object** `f`.
https://docs.python.org/3/glossary.html#term-file-object

```
1 f.readline()
```

```
'This is a demo file.\n'
```

Provides a method for reading a single line from the file. The string `'\n'` is a **special character** that represents a new line. More on this soon.

Reading files

```
keith@Steinhaus:~/demo$ cat demo.txt
This is a demo file.
It is a text file, containing three lines of text.
Here is the third line.
keith@Steinhaus:~/demo$
```

```
1 f = open('demo.txt')
2 f.readline()
```

```
'This is a demo file.\n'
```

```
1 f.readline()
```

```
'It is a text file, containing three lines of text.\n'
```

```
1 f.readline()
```

```
'Here is the third line.\n'
```

```
1 f.readline()
```

```
''
```

Each time we call `f.readline()`, we get the next line of the file...

...until there are no more lines to read, at which point the `readline()` method returns the empty string whenever it is called.

Reading files

```
1 f = open('demo.txt')
2 for line in f:
3     for wd in line.split():
4         print(wd.strip('.,'))
```

```
This
is
a
demo
file
It
is
a
text
file
containing
three
lines
of
text
Here
is
the
third
line
```

We can treat `f` as an iterator, in which each iteration gives us a line of the file.

Iterate over each word in the line (splitting on `' '` by default).

Remove the trailing punctuation from the words of the file.

`open()` provides a bunch more (optional) arguments, some of which we'll discuss later.
https://docs.python.org/3/library/functions.html#open

Reading files

```
1 with open('demo.txt') as f:
2     for line in f:
3         for wd in line.split():
4             print(wd.strip('.,'))
```

```
This
is
a
demo
file
It
is
a
text
file
containing
three
lines
of
text
Here
is
the
third
line
```

You may often see code written this way, using the `with` keyword. Don't worry about it for now; we'll see it in detail later. For now, it suffices to know that this is equivalent to what we did on the previous slide.

From the documentation: "It is good practice to use the with keyword when dealing with file objects. The advantage is that the file is properly closed after its suite finishes, even if an exception is raised at some point." https://docs.python.org/3/reference/compound_stmts.html#with

In plain English: the `with` keyword does a bunch of error checking and cleanup for you, automatically.

Writing files

Open the file in **write** mode. If the file already exists, this creates it anew, deleting its old contents.

```
1 f = open('animals.txt', 'w')
2 f.read()
```

If I try to read a file in write mode, I get an error.

```
---------------------------------------------------------------------------
UnsupportedOperation                                 Traceback (most recent call last)
<ipython-input-29-3b1ef477003a> in <module>()
      1 f = open('animals.txt', 'w')
----> 2 f.read()

UnsupportedOperation: not readable
```

Write to the file. This method returns the number of characters written to the file. Note that `'\n'` counts as a single character, the new line.

```
1 f.write('cat\n')
2 f.write('dog\n')
3 f.write('bird\n')
4 f.write('goat\n')
```

```
5
```

Writing files

```
1 f = open('animals.txt', 'w')
2 f.write('cat\n')
3 f.write('dog\n')
4 f.write('bird\n')
5 f.write('goat\n')
6 f.close()
```

Open the file in **write** mode.
This overwrites the version of the file created in the previous slide.

Each write appends to the end of the file.

When we're done, we close the file. This happens automatically when the program ends, but its good practice to close the file as soon as you're done.

```
1 f = open('animals.txt', 'r')
2 for line in f:
3     print(line, end="")
```

```
cat
dog
bird
goat
```

Now, when I open the file for reading, I can print out the lines one by one.

The lines of the file already include newlines on the ends, so override Python's default behavior of printing a newline after each line.

Aside: Formatting Strings

```
1 x = 23
2 print('x = %d' % x)
```

```
x = 23
```

```
1 animal = 'unicorn'
2 print('My pet %s' % animal)
```

```
My pet unicorn
```

```
1 x = 2.718; y = 1.618
2 print('%f divided by %f is %f' % (x,y,x/y))
```

```
2.718000 divided by 1.618000 is 1.679852
```

```
1 print('%.3f divided by %.3f is %.8f' % (x,y,x/y))
```

```
2.718 divided by 1.618 is 1.67985167
```

Python provides tools for formatting strings. Example: easier way to print an integer as a string.

`%d` : integer
`%s` : string
`%f` : floating point
More information:
https://docs.python.org/3/library/stdtypes.html#printf-style-string-formatting

Can further control details of formatting, such as number of significant figures in printing floats.

Newer features for similar functionality:
https://docs.python.org/3/reference/lexical_analysis.html#f-strings
https://docs.python.org/3/library/stdtypes.html#str.format

Aside: Formatting Strings

Note: Number of formatting arguments must match the length of the supplied tuple!

```
1 x = 2.718; y = 1.618
2 print('%f divided by %f is %f' % (x,y,x/y,1.0))
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-46-eb736fce3612> in <module>()
      1 x = 2.718; y = 1.618
----> 2 print('%f divided by %f is %f' % (x,y,x/y,1.0))

TypeError: not all arguments converted during string formatting
```

```
1 x = 2.718; y = 1.618
2 print('%f divided by %f is %f' % (x,y))
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-47-b2e6a26d3415> in <module>()
      1 x = 2.718; y = 1.618
----> 2 print('%f divided by %f is %f' % (x,y))

TypeError: not enough arguments for format string
```

Saving objects to files: `pickle`

Sometimes it is useful to be able to turn an object into a string

```
1 import pickle
2 t1 = [1,'two',3.0]
3 s = pickle.dumps(t1)
4 s
```

`pickle.dumps()` (short for “dump string”) creates a **binary string** representing an object.

```
b'\x80\x03]q\x00(K\x01X\x03\x00\x00\x00twoq\x01G@\x08\x00\x00\x00\x00\x00\x00e.'
```

This is a raw binary string that encodes the list `t1`. Each symbol encodes one byte. More detail later in the course.
https://docs.python.org/3.6/library/functions.html#func-bytes
https://en.wikipedia.org/wiki/ASCII

```
1 t2 = pickle.loads(s)
2 t1==t2
```

```
True
```

```
1 t1 is t2
```

```
False
```

Saving objects to files: `pickle`

Sometimes it is useful to be able to turn an object into a string

```
1 import pickle
2 t1 = [1,'two',3.0]
3 s = pickle.dumps(t1)
4 s
```

```
b'\x80\x03]q\x00(K\x01X\x03\x00\x00\x00twoq\x01G@\x08\x00\x00\x00\x00\x00\x00e.'
```

We can now use this string to store (a representation of) the list referenced by `t1`. We can write it to a file for later reuse, use it as a key in a dictionary, etc.

```
1 t2 = pickle.loads(s)
2 t1==t2
```

```
True
```

Later on, to “unpickle” the string and turn it back into an object, we use `pickle.loads()` (short for “load string”).

Important point: pickling stores a representation of the value, not the variable! So after this assignment, `t1` and `t2` are equivalent...

```
1 t1 is t2
```

```
False
```

...but not identical.

Locating files: the `os` module

```
1 import os
2 cwd = os.getcwd()
3 cwd
```

```
'/Users/keith/demo/L6_Files'
```

`os` module lets us interact with the operating system.
https://docs.python.org/3.6/library/os.html

`os.getcwd()` returns a string corresponding to the **current working directory**.

```
1 os.listdir()
```

```
['data', 'scripts']
```

```
1 os.listdir('data')
```

```
['numbers.txt', 'pi.txt']
```

`os.listdir()` lists the contents of its argument, or the current directory if no argument.

```
1 os.chdir('data')
2 os.getcwd()
```

```
'/Users/keith/demo/L6_Files/data'
```

`os.chdir()` changes the working directory. After calling `chdir()`, we're in a different cwd.

Locating files: the `os` module

```
1 import os
2 cwd = os.getcwd()
3 cwd
```

```
'/Users/keith/demo/L6_Files'
```

This is called a **path**. It starts at the **root directory**, `'/'`, and describes a sequence of nested directories.

```
1 os.listdir()
```

```
['data', 'scripts']
```

```
1 os.listdir('data')
```

```
['numbers.txt', 'pi.txt']
```

A path from the root to a file or directory is called an **absolute path**. A path from the current directory is called a **relative path**.

```
1 os.path.abspath('data/pi.txt')
```

```
'/Users/keith/demo/L6_Files/data/pi.txt'
```

Use `os.path.abspath` to get the absolute path to a file or directory.

Locating files: the `os` module

```
1 import os
2 os.chdir('/Users/keith/demo/L6_Files')
3 os.listdir('data')
```

```
['extra', 'numbers.txt', 'pi.txt']
```

```
1 os.path.exists('data/pi.txt')
```

```
True
```

```
1 os.path.exists('data/nonsense.txt')
```

```
False
```

Check whether or not a file/directory exists.

```
1 os.path.isdir('data/extra')
```

```
True
```

```
1 os.path.isdir('data/numbers.txt')
```

```
False
```

Check whether or not this is a directory. `os.path.isfile()` works analogously.

Handling errors: try/catch statements

Sometimes when an error occurs, we want to try and recover

Rather than just giving up and having Python yell at us.

Python has a special syntax for this: `try:... except:...`

Basic idea: try to do something, and if an error occurs, try something else.

Example: try to open a file for reading.

If that fails (e.g., because the file doesn't exist) look for the file elsewhere

Handling errors: try/catch statements

```
1 import os
2 os.listdir()
```

```
['backup_file.txt', 'data', 'scripts']
```

```
1 try:
2     f = open('nonsense.txt')
3 except:
4     f = open('backup_file.txt')
5 f.read()
```

```
'This is a backup file.\n'
```

Python attempts to execute the code in the `try` block. If that runs successfully, then we continue on.

If the `try` block fails (i.e., if there's an **exception**), then we run the code in the `except` block.

Programmers call this kind of construction a **try/catch statement**, even though the Python syntax uses `try/except` instead.

Handling errors: try/catch statements

```
1  import math
2  def my_sqrt(x):
3      try:
4          return math.sqrt(x)
5      except TypeError:
6          print('Type does not allow sqrt.')
7      except ValueError:
8          print('Negative?')
9      except:
10         print('Gosh, no idea what happened.')
```

```
1  my_sqrt('cat')
```

```
Type does not allow sqrt.
```

```
1  my_sqrt(-10)
```

```
Negative?
```

Remember that `TypeError` means `x` was of a type that doesn't support `sqrt`.

`ValueError` means `x` was of valid type, but value doesn't make sense for the operation (Python module for complex math: `cmath`).

Note: we don't see an error raised. Here, we decided to print information, but it's more common to use try/catch to recover from the error.

Writing modules

Python provides modules (e.g., math, os, time)

But we can also write our own, and import from them with same syntax

```
1 import prime
2 prime.is_prime(2)
```

```
True
```

```
1 prime.is_prime(3)
```

```
True
```

```
1 prime.is_prime(1)
```

```
False
```

```
1 prime.is_prime(23)
```

```
True
```

prime.py

```
import math

def is_prime(n):
    if n <= 1:
        return False
    elif n==2:
        return True
    else:
        ulim = math.ceil(math.sqrt(n))
        for k in range(2,ulim+1):
            if n%k==0:
                return False
        return True
```

Writing modules

Import everything defined in `prime`, so we can call it without the prefix. Can also import specific functions: `from prime import is_square`

```
1 from prime import *
2 is_prime(7)
```

```
True
```

```
1 is_square(7)
```

```
False
```

```
1 is_prime(373)
```

```
True
```

Caution: be careful that you don't cause a collision with an existing function or a function in another module!

prime.py

```
1  import math
2
3  def is_prime(n):
4      if n <= 1:
5          return False
6      elif n==2:
7          return True
8      else:
9          ulim = math.ceil(math.sqrt(n))
10         for k in range(2,ulim+1):
11             if n%k==0:
12                 return False
13         return True
14 def is_square(n):
15     r = int(math.sqrt(n))
16     return(r*r==n or (r+1)*(r+1)==n)
```

Readings (this lecture)

Required:

Downey Chapter 14 or Severance Chapter 7

Python File I/O Documentation:

https://docs.python.org/3/tutorial/inputoutput.html

Handling Errors and Exceptions:

https://docs.python.org/3/tutorial/errors.html

Recommended:

Python pickle module:

https://docs.python.org/3/library/pickle.html#module-pickle

Readings (next lecture)

Required:

Downey Chapter 15

Python documentation on classes (only through section 9.3):

https://docs.python.org/3/tutorial/classes.html

Recommended:

D. Phillips (2015). *Python 3 Object-oriented Programming*, *Second Edition*. Packt Publishing.

M. Weisfeld (2009). *The Object-Oriented Thought Process, Third Edition*. Addison-Wesley.