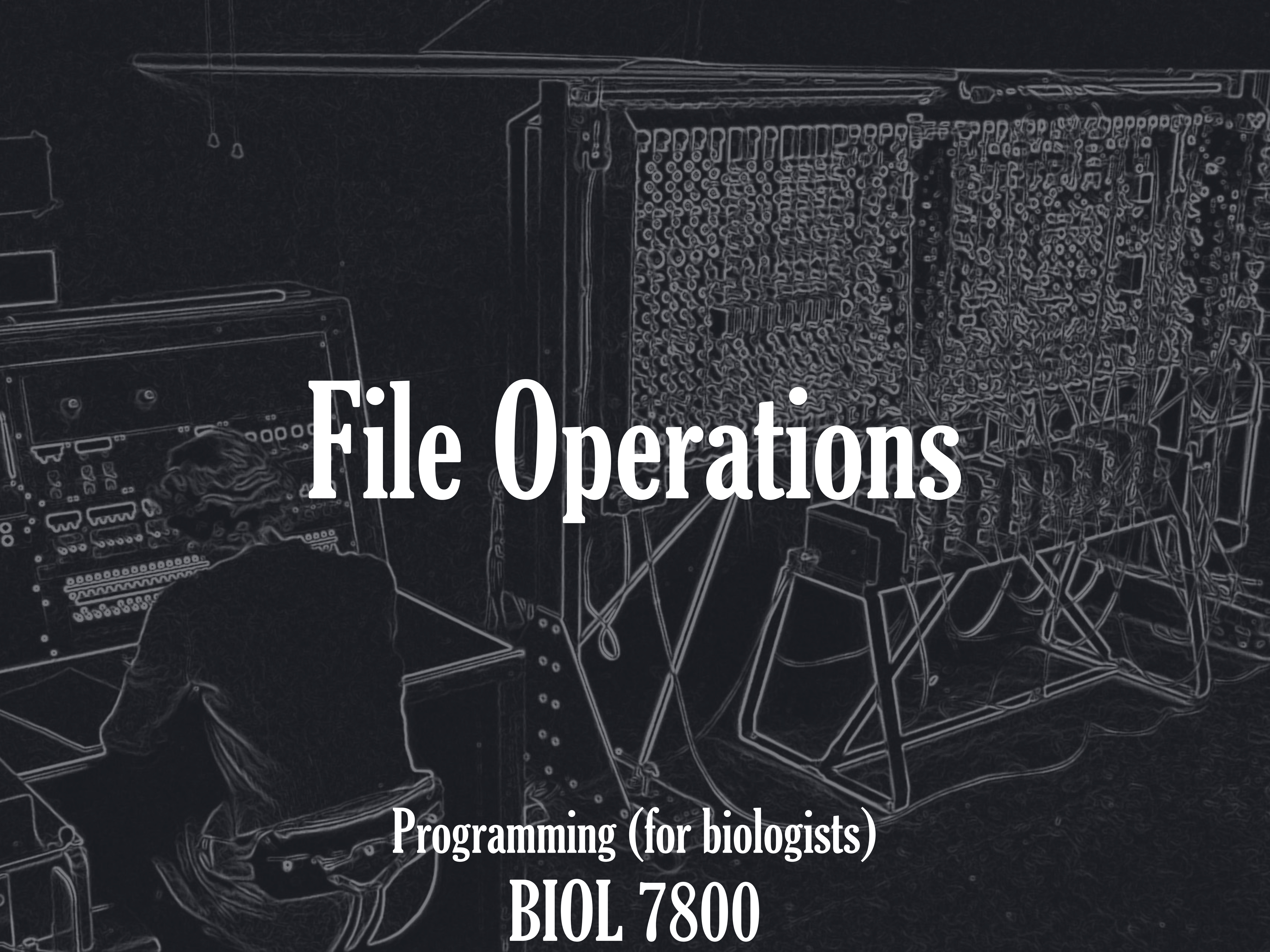


File Operations

Programming (for biologists)

BIOL 7800

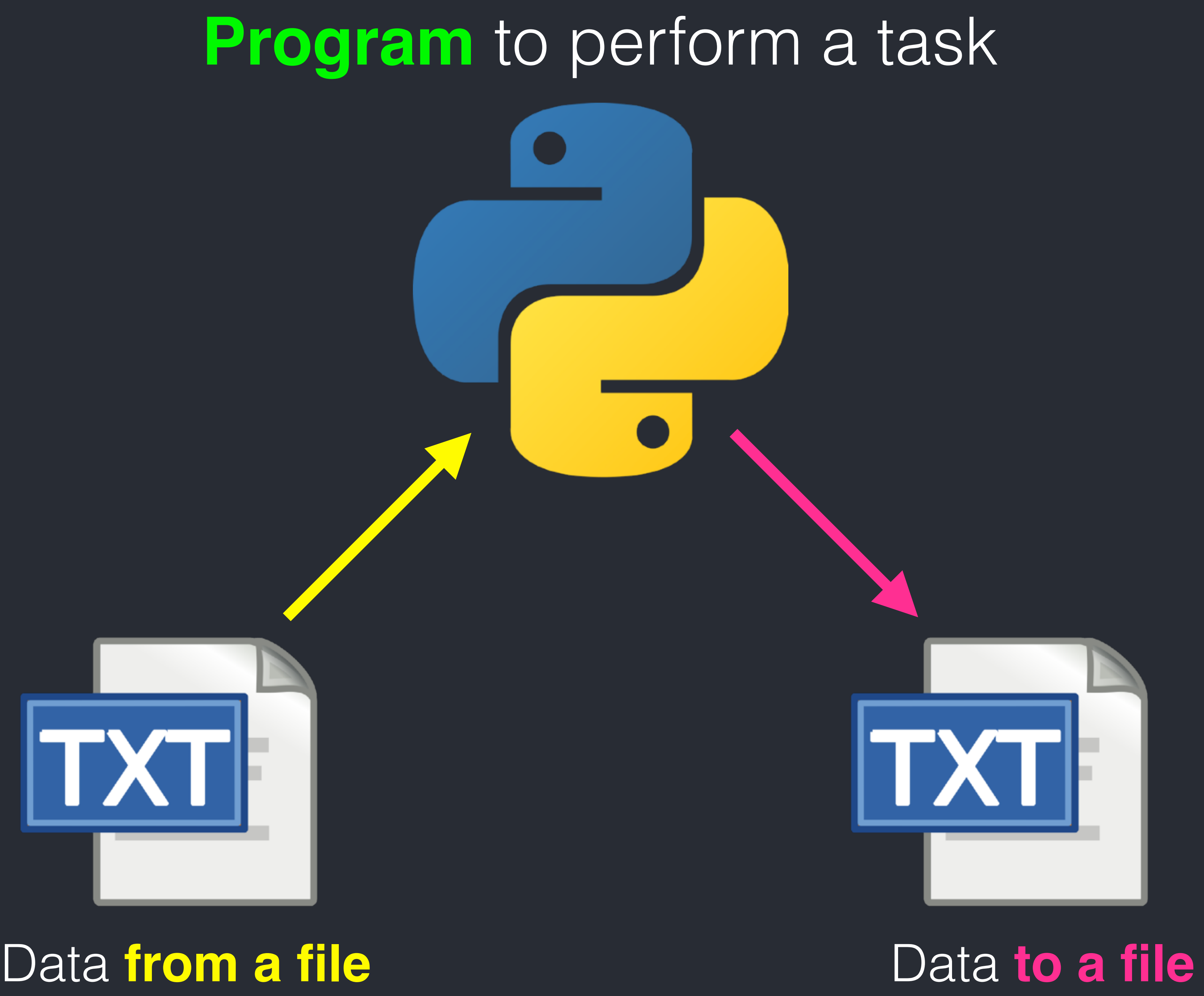
Up until now…

We've been using text from the body of our code.

Stars burn a very light type of air that is packed very tight in the middle of the star. The tiny pieces of the air join to form a slightly heavier type of air that can't burn in the star without the middle being much hotter and tighter.

After a long time, all the very light air in the middle is used up. Then the star gets much hotter and tighter in the middle, so it can burn the slightly heavier air. It also also gets much bigger and cooler outside. But it is still so hot it will burn up any close-in worlds.

But, we often want to…



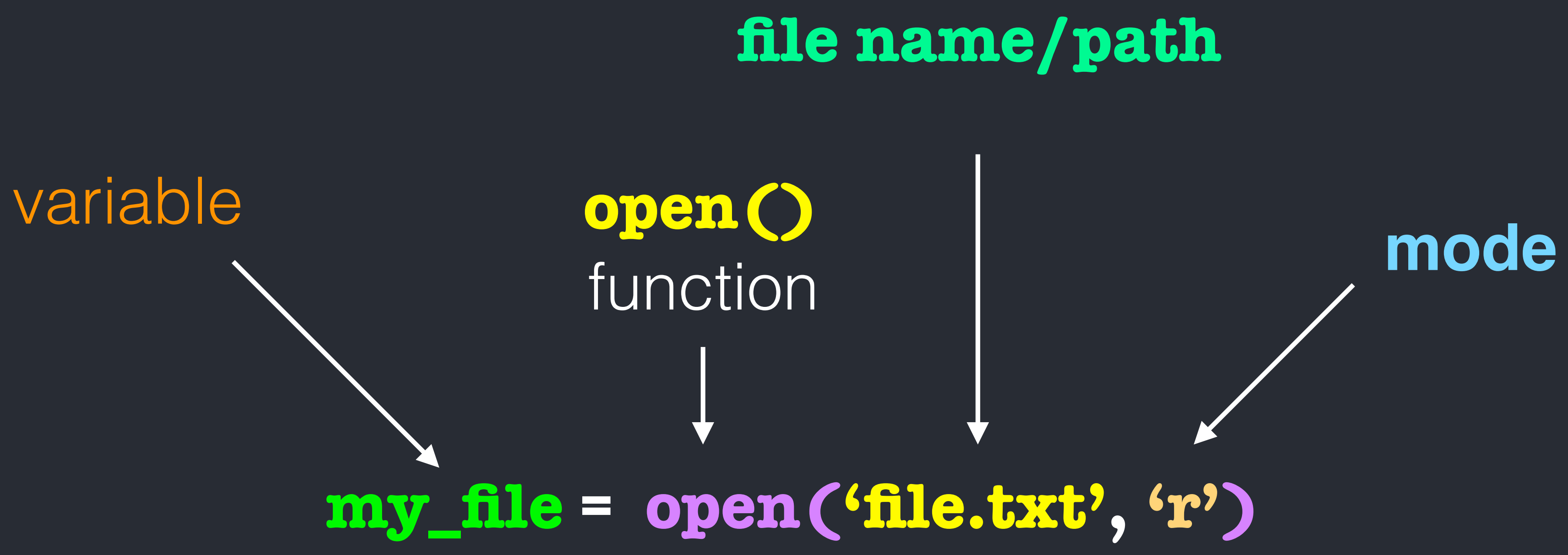
File Operations (AKA File IO)

The key to file operations is **`open()`**

And, in python, we need to **`open()`** a file before we can read from it or write to it

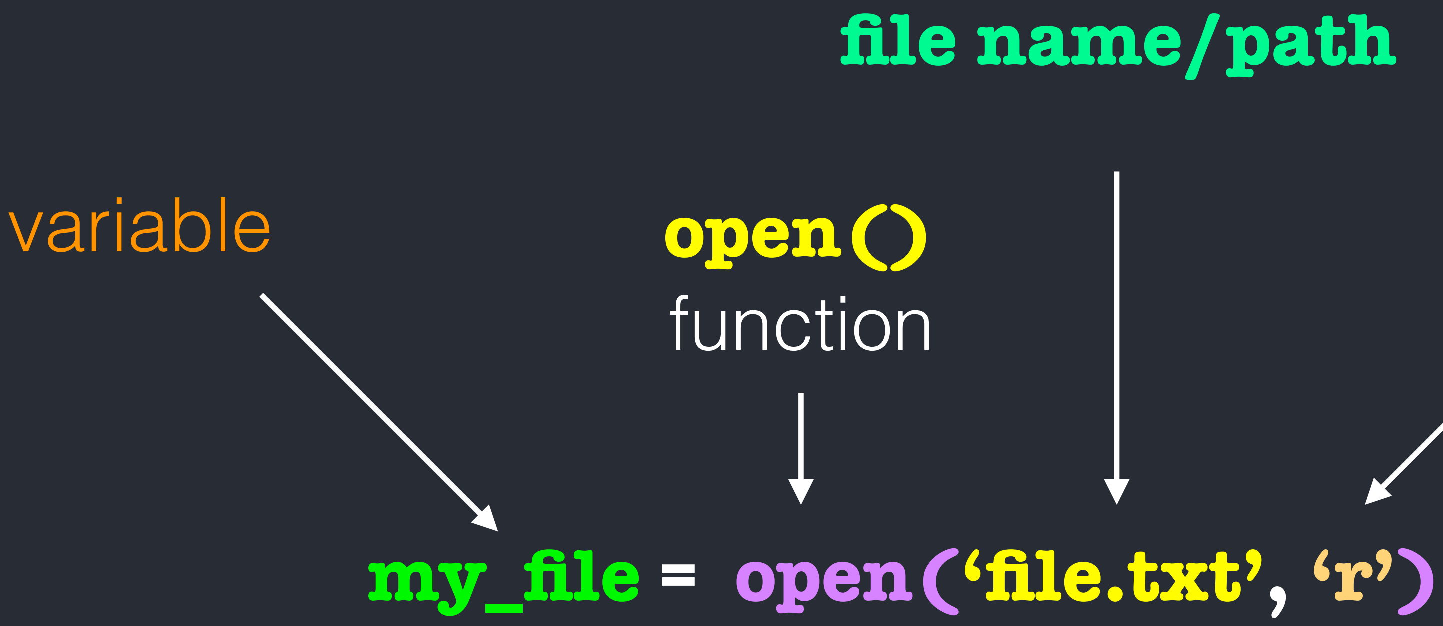
`open()` has several “forms”

Anatomy of open()



```
my_file = open('file.txt', 'r')
```

Anatomy of open()



```
my_file = open('file.txt', 'r')
```

"mode" comes in 3 flavors:

'r'	:: read
'w'	:: write
'a'	:: append

Anatomy of open()

"mode" comes in 3 flavors

`open('file.txt', 'r')`
read

For reading lines of/entire file

non-destructive

`open('file.txt', 'w')`
write

For writing lines to a file

destructive

Will **erase contents** of any existing file with name of file you open !!

`open('file.txt', 'a')`
append

For writing lines to end of a file

non-destructive

Anatomy of open()

"mode" comes in 3 flavors

```
open('file.txt', 'r+')
```

read + write

plus

For writing lines
to a file

For reading
lines of/entire file

non-destructive

Anatomy of open() [binary files]

"mode" can also be altered for operations on "binary" files (jpeg, tiff, docx, etc.)

(THIS IS MORE IMPORTANT ON **WINDOWS** THAN **UNIX**)

open('file.txt', 'rb')	open('file.txt', 'wb')	open('file.txt', 'ab')
read	write	append
For reading lines of/entire file	For writing lines to a file	For writing lines to end of a file
non-destructive	**destructive**	non-destructive

Anatomy of open()

All file modes

Character	Meaning
‘r’	open for reading
‘w’	open for writing, truncating file first
‘x’	open for exclusive creation, failing if the file already exists
‘a’	open for writing, appending to the end of the file if it exists
‘b’	binary mode
‘t’	text mode (default)
‘+’	open a disk file for updating (reading and writing)

Reading a file

```
my_file = open('file.txt', 'r')
```

returns a
file object

```
In:   my_file = open('file.txt', 'r')
In:   type(my_file)
Out: _io.TextIOWrapper
```

Reading a file

```
this is my file
i do not want it to be big
nor do i want it to be small
```

file.txt

```
my_file = open(‘file.txt’, ‘r’)
```

Reading a file

There are several **methods** you can use to read contents

.read()

```
In: my_file =  open('file.txt', 'r')
In: my_file.read()
Out: 'this is my file\ni do not want it to be big\nnor do i want it to be small\n'
```

The .read() method, reads the entire file into memory

Can use lots of RAM (when a huge file)

Reading a file

There are several **methods** you can use to read contents

.readline()

```
In:  my_file = open('file.txt', 'r')
In:  my_file.readline()
Out: this is my file\n
In:  my_file.readline()
Out: i do not want it to be big\n
In:   my_file.readline()
Out: nor do i want it to be small\n'
```

The .readline() method reads the file line-by-line

Only 1 line per call (inconvenient)

Reading a file

There are several **methods** you can use to read contents

.readlines()

```
In:  my_file = open(‘file.txt’, ‘r’)
In:   my_file.readlines()
Out: ['this is my file\n',
 'i do not want it to be big\n',
 'nor do i want it to be small\n']
```

The .readlines() method, reads the file into a list, splitting each line on the newline character to make a list entry

Can use lots of RAM (when a huge file)

Reading a file (the best way)

There are several methods you can use to read contents

Treat the file as an iterator

```
In:  my_file =  open('file.txt', 'r')
In:   for line in my_file
          # do something with line
          print(line.strip())
Out: this is my file
Out: i do not want it to be big
Out: nor do i want it to be small
```

⟵ we can just iterate over each line

Uses very little RAM !

Give us easy-access to entire file (line by line) !

Closing a file

```
In:   my_file = open('file.txt', 'r')
In:   # do stuff
```

When we .open() a file, we need to .close() it once we're done using it

```
In:   my_file.close()
```

This (1) helps avoid file corruption issues and (2) also helps remove stale links to different files

Reading a file with with

but all this .open() and .close() is bothersome

We can use with to accomplish both tasks

with helps us open file and access it using my_file

```
In:  with open('file.txt', 'r') as my_file:
In:      for line in my_file:
             print(line.strip())
```

with also closes file when we finish iterating over its line

Writing a file

Very similar to .read() on a file object… but using .write()

```
In:   my_text = “this is my file\n
i do not want it to be big\n
nor do i want it to be small”

In:   my_file = open(‘file.txt’, ‘w’)
In:   my_file.write(my_text)
Out: my_file.close()
```

open new file in write mode

use the write method to
write a line to the file

use the .close() method to
close the file

Writing a file

.writelines() is the writing corollary of .readlines()

A list of strings

↓

```
In:  my_lines = [“this\n”, “that\n”, “the other\n”]
In:  my_file =  open(‘file.txt’, ‘w’)
In:  my_file.writeline(my_lines)
In:  my_file.close()
```

Writing a file with with

again, all this .open() and .close() is bothersome

We can use with to accomplish both tasks

```
In:  my_text = "this is my file\n
i do not want it to be big\n
nor do i want it to be small"
```

with helps us open file and access it using my_file

```
In:  with open('file.txt', 'w') as my_file:
In:      my_file.write(my_text)
```

with also closes file when we finish iterating over its line

with to read and write

What does this do?

```
this is my file
i do not want it to be big
nor do i want it to be small
```

input.txt

```
In:  with open(‘input.txt’, ‘r’) as my_input:
In:      with open(‘output.txt’, ‘w’) as my_output:
In:          for line in my_input:
                 my_output.write(line)
```

Formatting what you write

Up to now, you've been using the print() function

What I want:

“print, something, like, this”

What you usually do:

```
print(“print, something, like, this”)
print(‘ print,’ + ‘ something,’ + ‘ like,’ + ‘ this’)
```

Introducing format()

What I want:

“print, something, like, this”

Using the .format() string method

```
print('{0}, {1}, {2}, {3}'.format('print', 'something', 'like', ‘this'))
```

Strings in parens gets substituted to the indexed { }

Introducing format()

What I want:

“print, something, like, this”

Using the .format() string method

```
print('{ }, { }, { }, { }'.format('print', 'something', 'like', ‘this'))
```

We can leave out the index numbers and

strings in parens gets substituted to their relative { }

Introducing format()

What I want:

"print, something, like, this"

Using the .format() string method

```
print('{}, {}, {}, {}'.format('print', 'something', 'like',))
```

What do you think happens above?

Introducing format()

We can also repeat indexes to repeat a word…

(but we **must** give index position in this case)

Using the .format() string method

```
print('{0}, {1}, {2}, {3}, {1}’.format(‘some’, 'dogs', 'like', 'other'))
```

What it prints:

“some dogs like other dogs”

Introducing format()

The % (format) operator is another way to do string substitution

But, the .format() method is much more powerful

```
camels = 124
print(“I have seen %d camels” % camels)
```

vs.

```
camels = 124
print(“I have seen {0} camels”.format(camels))
```