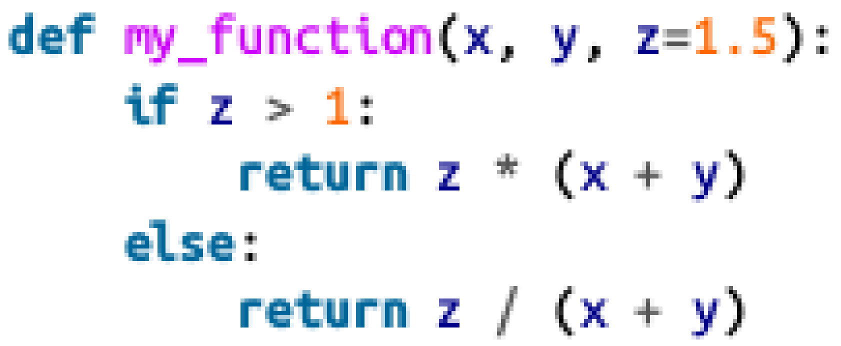
# Python Basics

*for Data Analysis*

By Armin Khayati

# Python Basics : Functions

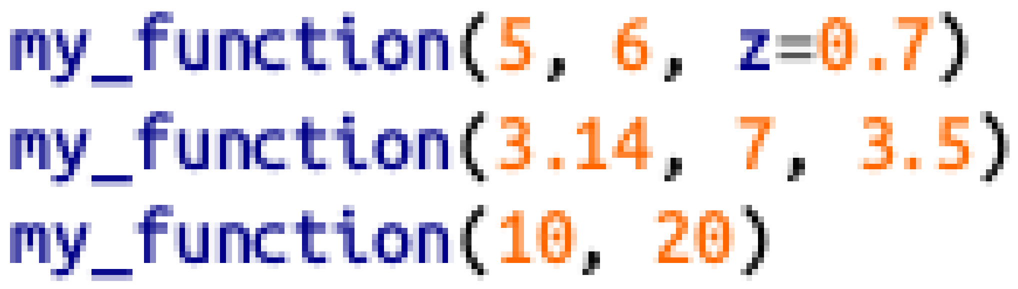
- Functions are the primary and most important method of code organization and reuse in Python.
- Functions are declared with the `def` keyword and returned from with the return keyword.



```python
def my_function(x, y, z=1.5):
    if z > 1:
        return z * (x + y)
    else:
        return z / (x + y)
```

# Python Basics : Functions

- There is no issue with having multiple return statements.
- If Python reaches the end of a function without encountering a return statement, `None` is returned automatically.
- Each function can have positional arguments and keyword arguments.

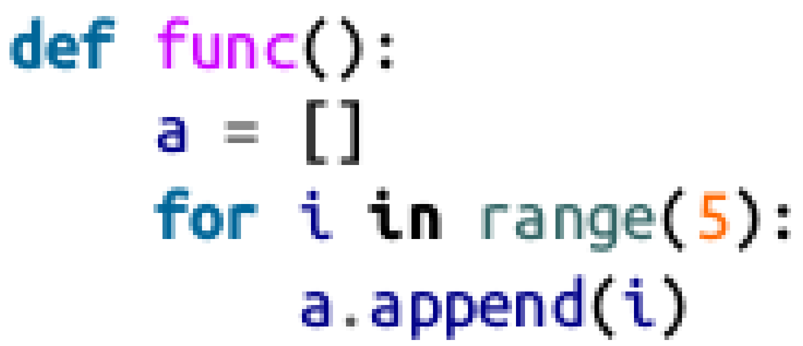


```
my_function(5, 6, z=0.7)
my_function(3.14, 7, 3.5)
my_function(10, 20)
```

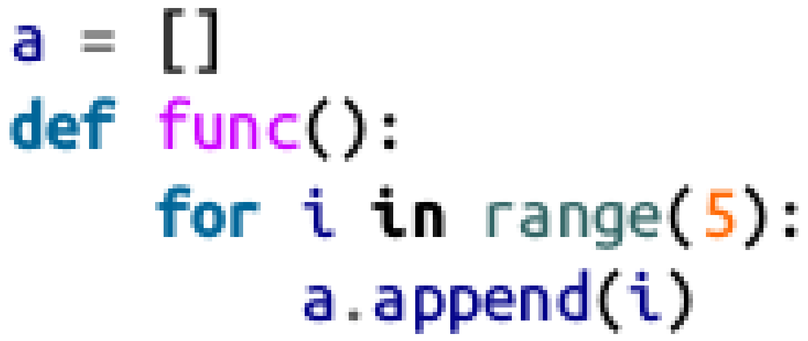
# Python Basics : Namespaces, Scope, and Local Functions

- Functions can access variables in two different scopes: `global` and `local`.
- An alternative and more descriptive name describing a variable scope in Python is a `namespace`.
- Any variables that are assigned within a function by default are assigned to the local namespace.
- The local namespace is created when the function is called and immediately populated by the function’s arguments. After the function is finished, the local namespace is destroyed.

# Python Basics : Namespaces, Scope, and Local Functions



```
def func():
    a = []
    for i in range(5):
        a.append(i)
```



```
a = []
def func():
    for i in range(5):
        a.append(i)
```

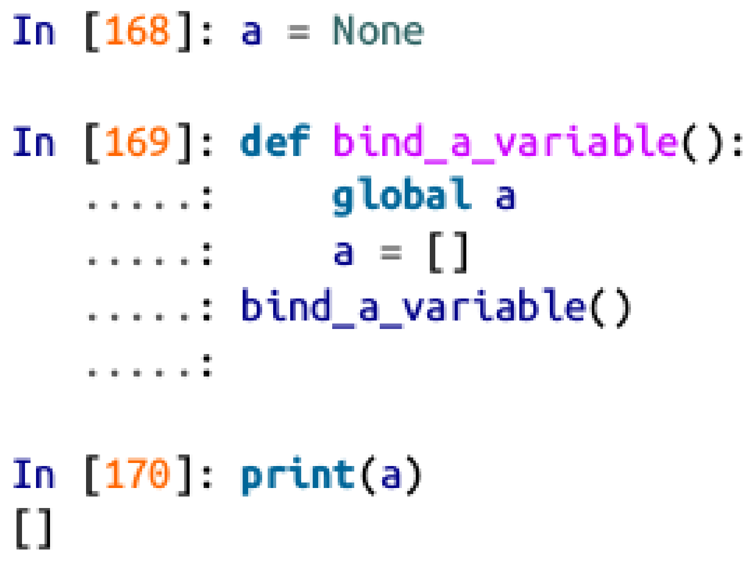
**Difference???**

## Python Basics : Namespaces, Scope, and Local Functions

- First one : When `func()` is called, the empty list `a` is created, five elements are appended, and then `a` is destroyed when the function exits.
- Second one : When `func()` is called, five elements are appended, and the list `a` still exists when the function exits.

# Python Basics : Namespaces, Scope, and Local Functions

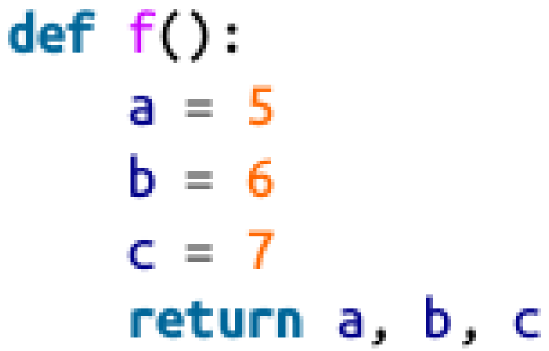
- Assigning variables outside of the function's scope is possible, but those variables must be declared as global via the `global` keyword.



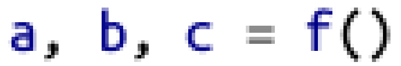
```
In [168]: a = None

In [169]: def bind_a_variable():
   .....:     global a
   .....:     a = []
   .....: bind_a_variable()
   .....:

In [170]: print(a)
[]
```

# Python Basics : Returning Multiple Values

- Here’s an example:

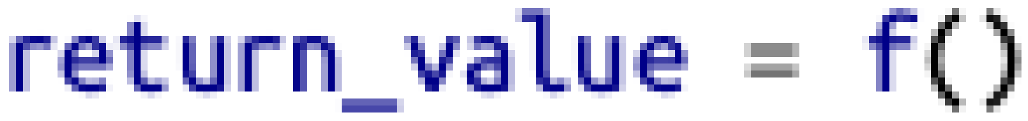


```
def f():
    a = 5
    b = 6
    c = 7
    return a, b, c
```



```
a, b, c = f()
```

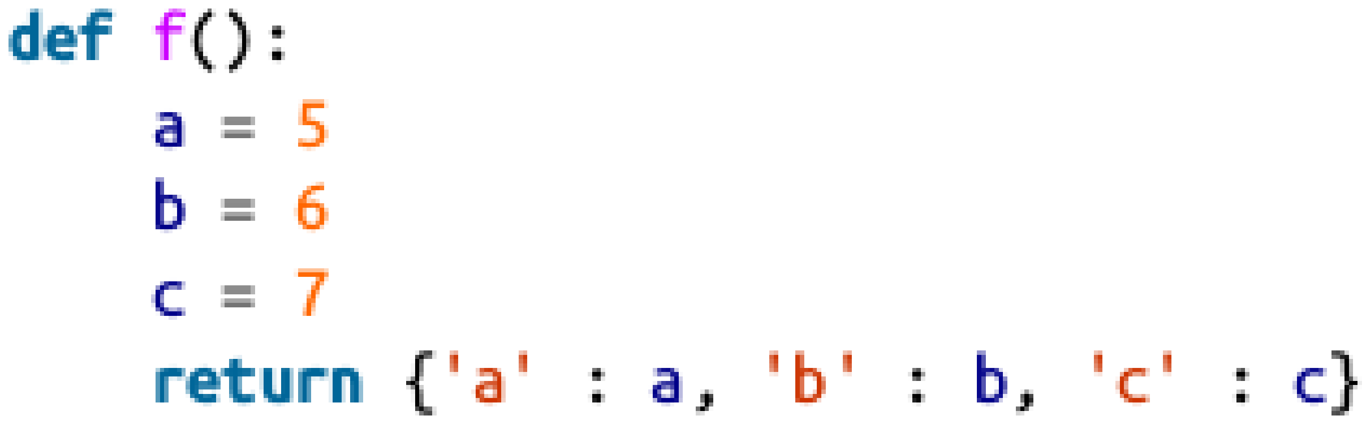
- What’s happening here is that the function is actually just returning one object, namely a tuple, which is then being unpacked into the result variables.
- We could have done this instead



```
return_value = f()
```

# Python Basics : Returning Multiple Values

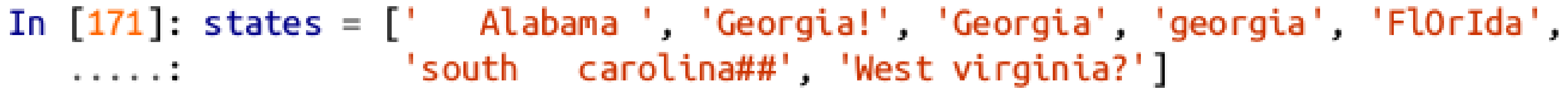
- A potentially attractive alternative to returning multiple values like before might be to return a `dict` instead.



```
def f():
    a = 5
    b = 6
    c = 7
    return {'a' : a, 'b' : b, 'c' : c}
```

# Python Basics : Functions Are Objects

- Since Python functions are objects, many constructs can be easily expressed that are difficult to do in other languages.
- Suppose we were doing some data cleaning and needed to apply a bunch of transformations to the following list of strings.

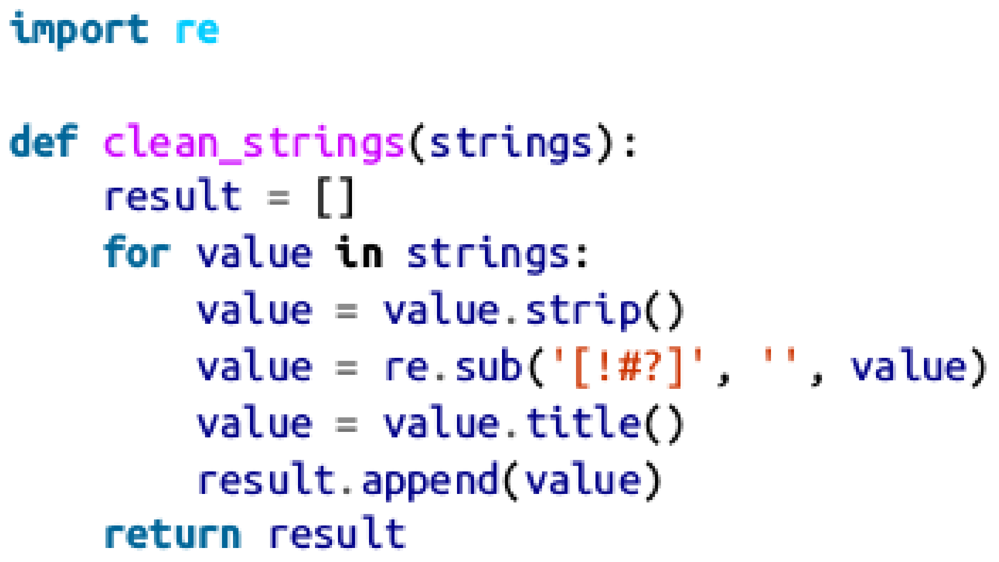


```
In [171]: states = ['   Alabama ', 'Georgia!', 'Georgia', 'georgia', 'FlOrIda',
   .....:           'south   carolina##', 'West virginia?']
```

- Lots of things need to happen to make this list of strings uniform and ready for analysis: stripping whitespace, removing punctuation symbols, and standardizing on proper capitalization.

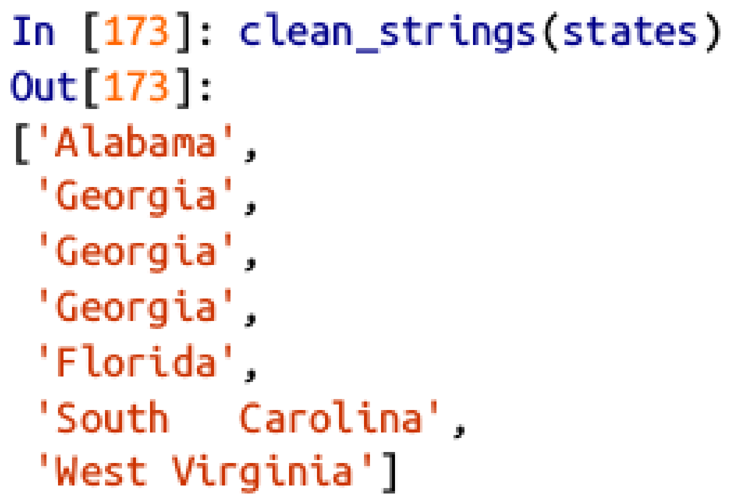
# Python Basics : Functions Are Objects

- One way to do this is to use built-in string methods along with the re standard library module for regular expressions.



```python
import re

def clean_strings(strings):
    result = []
    for value in strings:
        value = value.strip()
        value = re.sub('[!#?]', '', value)
        value = value.title()
        result.append(value)
    return result
```

- The result looks like this:



```
In [173]: clean_strings(states)
Out[173]:
['Alabama',
 'Georgia',
 'Georgia',
 'Georgia',
 'Florida',
 'South   Carolina',
 'West Virginia']
```

# Python Basics : Functions Are Objects

- An alternative approach that you may find useful is to make a list of the operations you want to apply to a particular set of strings.

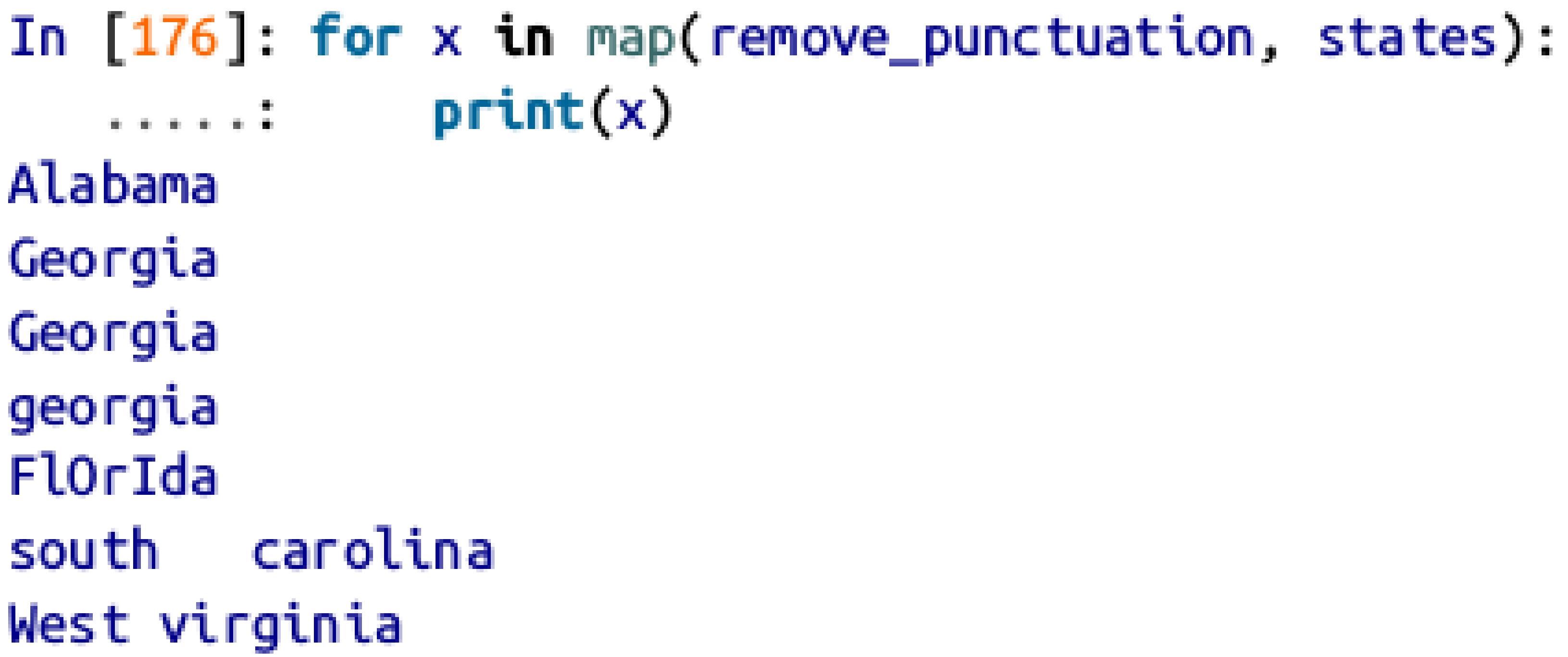
```python
def remove_punctuation(value):
    return re.sub('[!#?]', '', value)

clean_ops = [str.strip, remove_punctuation, str.title]

def clean_strings(strings, ops):
    result = []
    for value in strings:
        for function in ops:
            value = function(value)
        result.append(value)
    return result
```

- Then we have the following:

```
In [175]: clean_strings(states, clean_ops)
Out[175]:
['Alabama',
 'Georgia',
 'Georgia',
 'Georgia',
 'Florida',
 'South   Carolina',
 'West Virginia']
```

# Python Basics : Functions Are Objects

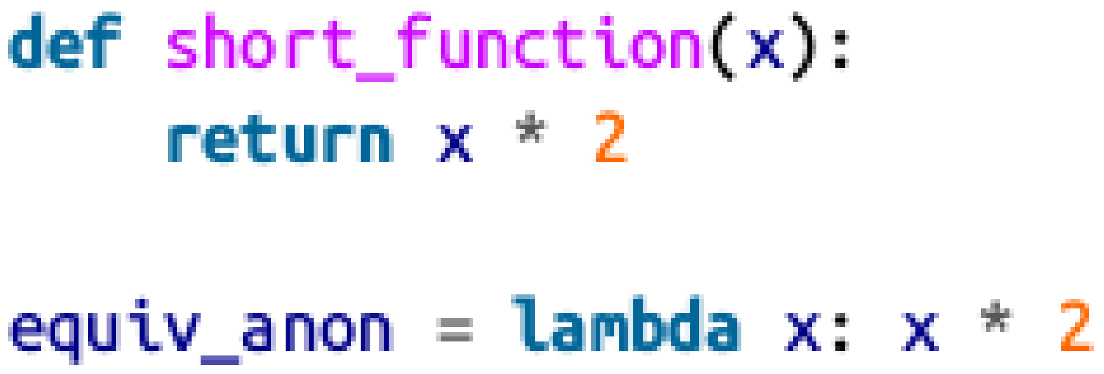
- A more functional pattern like this enables you to easily modify how the strings are transformed at a very high level. The `clean_strings` function is also now more reusable and generic.
- You can use functions as arguments to other functions like the built-in map function, which applies a function to a sequence of some kind:
- .



```
In [176]: for x in map(remove_punctuation, states):
   .....:     print(x)
Alabama
Georgia
Georgia
georgia
FlOrIda
south   carolina
West virginia
```

# Python Basics : Anonymous (Lambda) Functions

- Python has support for so-called anonymous or lambda functions, which are a way of writing functions consisting of a single statement, the result of which is the return value.
- They are defined with the lambda keyword, which has no meaning other than “we are declaring an anonymous function”.



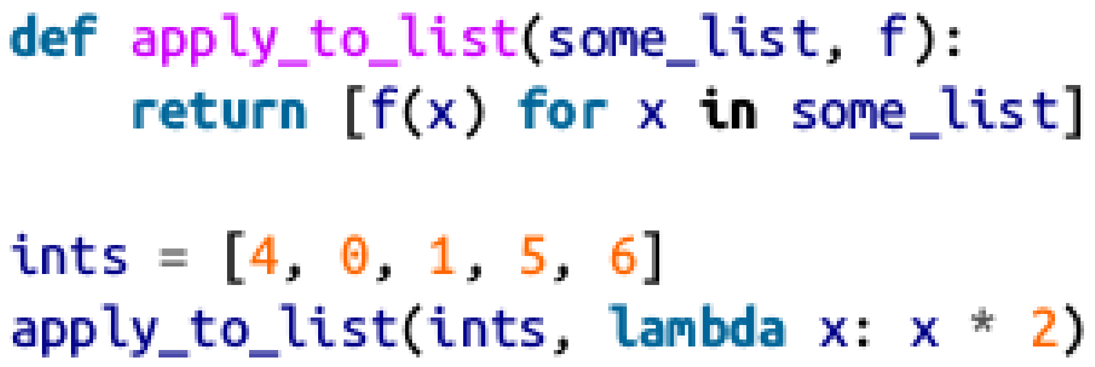
```python
def short_function(x):
    return x * 2

equiv_anon = lambda x: x * 2
```

## Python Basics : Anonymous (Lambda) Functions

- They are especially convenient in data analysis because, as you’ll see, there are many cases where data transformation functions will take functions as arguments.
- It’s often less typing (and clearer) to pass a lambda function as opposed to writing a full-out function declaration or even assigning the lambda function to a local variable.

# Python Basics : Anonymous (Lambda) Functions

- For example, consider this silly example:



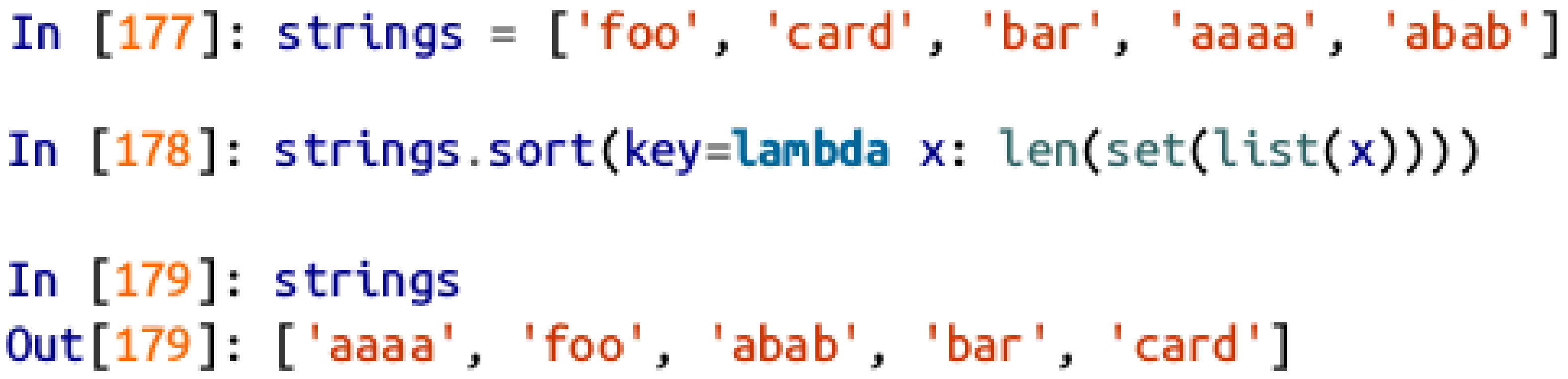
```python
def apply_to_list(some_list, f):
    return [f(x) for x in some_list]

ints = [4, 0, 1, 5, 6]
apply_to_list(ints, lambda x: x * 2)
```

- You could also have written

  `[x * 2 for x in ints]` , but here we were able to succinctly pass a custom operator to the `apply_to_list` function.

# Python Basics : Anonymous (Lambda) Functions

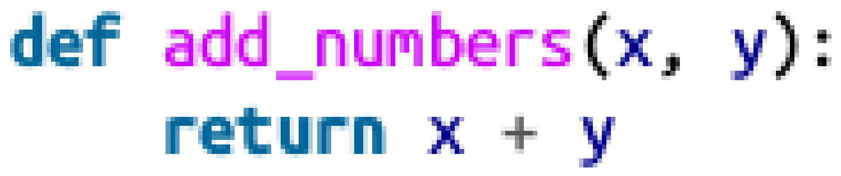
- As another example, suppose you wanted to sort a collection of strings by the number of distinct letters in each string:



```
In [177]: strings = ['foo', 'card', 'bar', 'aaaa', 'abab']

In [178]: strings.sort(key=lambda x: len(set(list(x))))

In [179]: strings
Out[179]: ['aaaa', 'foo', 'abab', 'bar', 'card']
```

# Python Basics
# Currying : Partial Argument Application

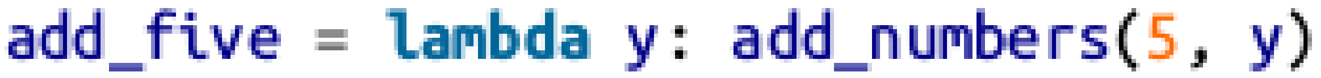
- Currying is computer science jargon (named after the mathematician Haskell Curry) that means deriving new functions from existing ones by partial argument application.
- For example, suppose we had a trivial function that adds two numbers together.



```
def add_numbers(x, y):
    return x + y
```

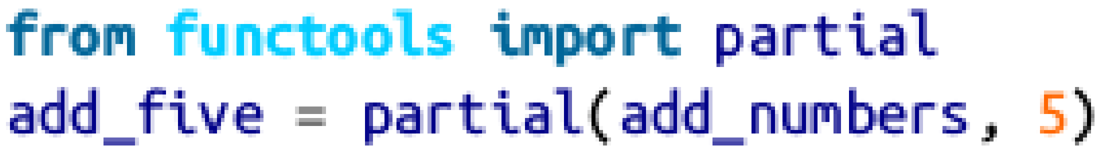
# Python Basics
# Currying : Partial Argument Application

- Using this function, we could derive a new function of one variable, add_five , that adds 5 to its argument:



```
add_five = lambda y: add_numbers(5, y)
```

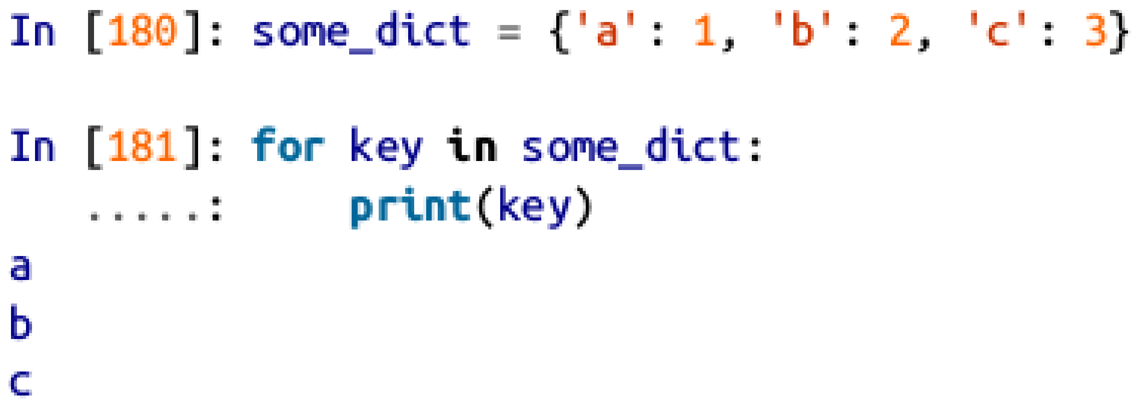
- The second argument to add_numbers is said to be curried.
- The built-in `functools` module can simplify this process using the partial function:



```
from functools import partial
add_five = partial(add_numbers, 5)
```

# Python Basics : Generators

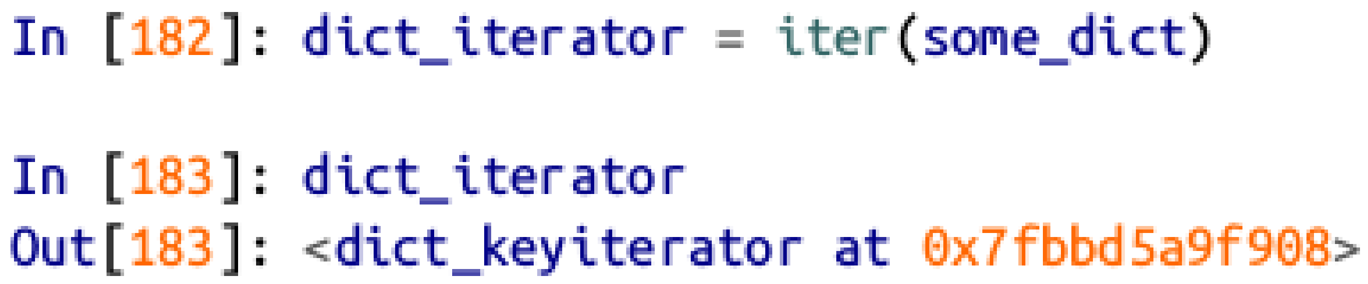
- When you write `for key in some_dict`, the Python interpreter first attempts to create an iterator out of some_dict



```
In [180]: some_dict = {'a': 1, 'b': 2, 'c': 3}

In [181]: for key in some_dict:
   .....:     print(key)
a
b
c
```

# Python Basics : Generators

- An iterator is any object that will yield objects to the Python interpreter when used in a context like a for loop.

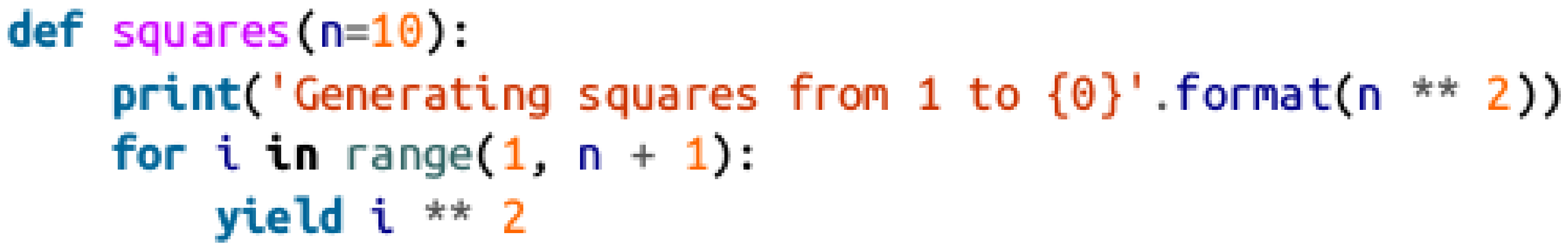


```
In [182]: dict_iterator = iter(some_dict)

In [183]: dict_iterator
Out[183]: <dict_keyiterator at 0x7fbbd5a9f908>
```

## Python Basics : Generators

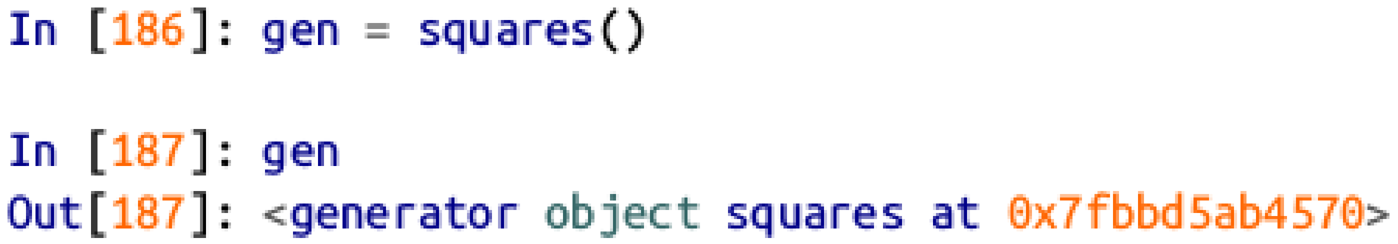
- A generator is a concise way to construct a new iterable object.
- Normal functions execute and return a single result at a time.
- Generators return a sequence of multiple results lazily, pausing after each one until the next one is requested.
- To create a generator, use the `yield` keyword instead of return in a function.

# Python Basics : Generators



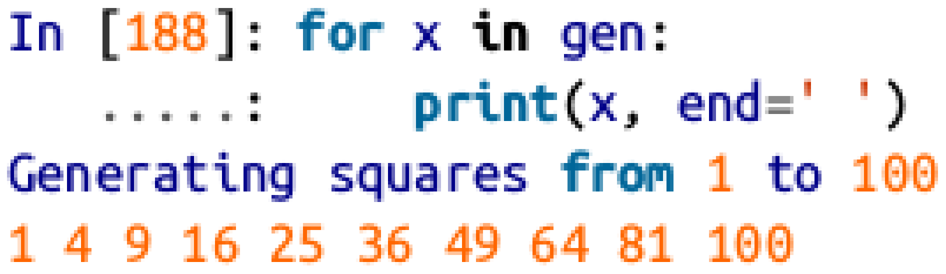
```
def squares(n=10):
    print('Generating squares from 1 to {0}'.format(n ** 2))
    for i in range(1, n + 1):
        yield i ** 2
```

- When you actually call the generator, no code is immediately executed.



```
In [186]: gen = squares()

In [187]: gen
Out[187]: <generator object squares at 0x7fbbd5ab4570>
```

- It is not until you request elements from the generator that it begins executing its code



```
In [188]: for x in gen:
   .....:     print(x, end=' ')
Generating squares from 1 to 100
1 4 9 16 25 36 49 64 81 100
```

# Python Basics : Generator expresssions

- Another even more concise way to make a generator is by using a generator expression.

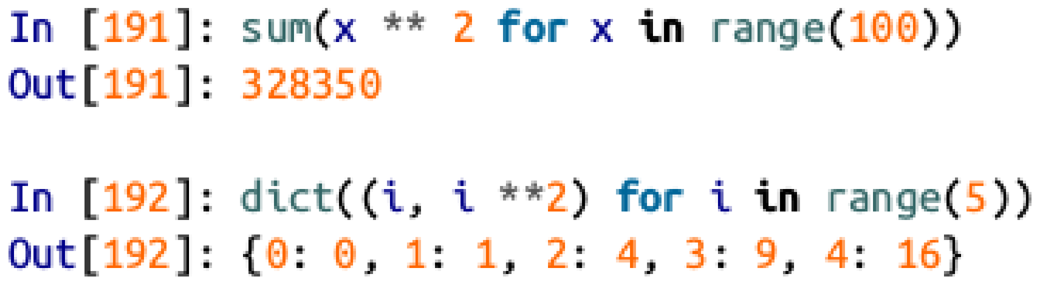
```
In [189]: gen = (x ** 2 for x in range(100))

In [190]: gen
Out[190]: <generator object <genexpr> at 0x7fbbd5ab29e8>
```

- This is completely equivalent to the following more verbose generator:

```
def _make_gen():
    for x in range(100):
        yield x ** 2
gen = _make_gen()
```

# Python Basics :
# Generator expresssions

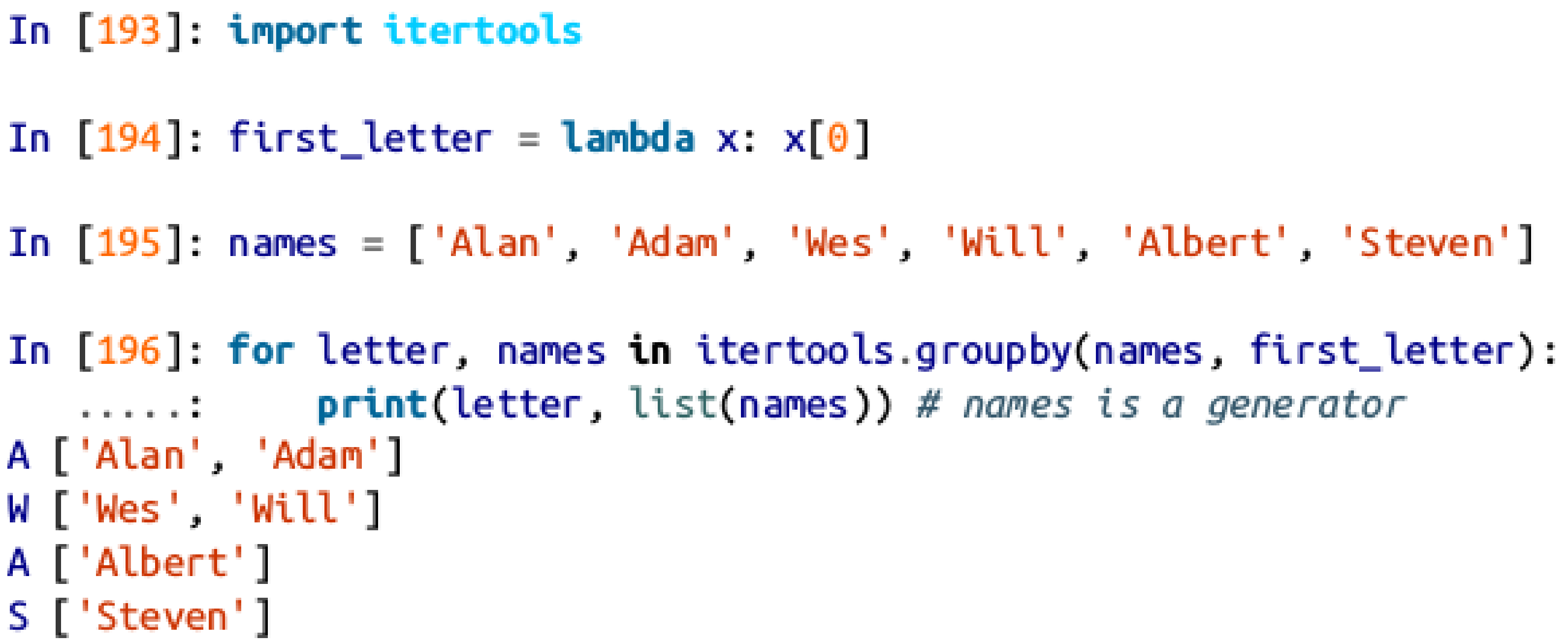
- Generator expressions can be used instead of list comprehensions as function arguments in many cases.



```
In [191]: sum(x ** 2 for x in range(100))
Out[191]: 328350

In [192]: dict((i, i **2) for i in range(5))
Out[192]: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

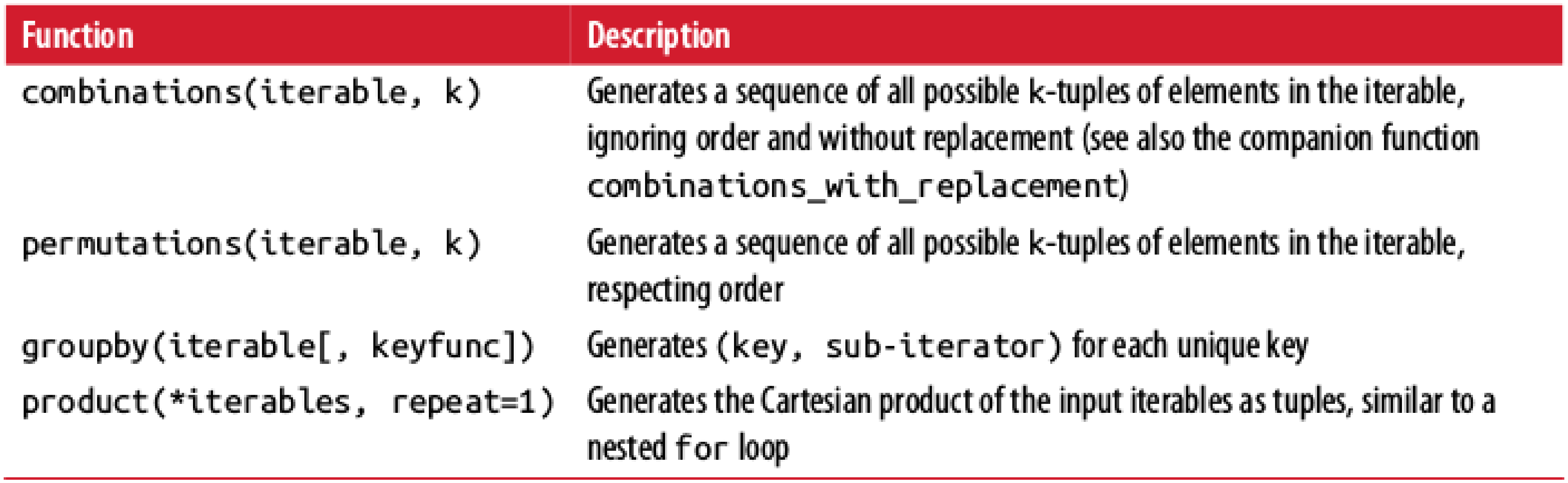
## Python Basics : Itertools module

- The standard library `itertools` module has a collection of generators for many common data algorithms.
- For example, `groupby` takes any sequence and a function, grouping consecutive elements in the sequence by return value of the function.
- .



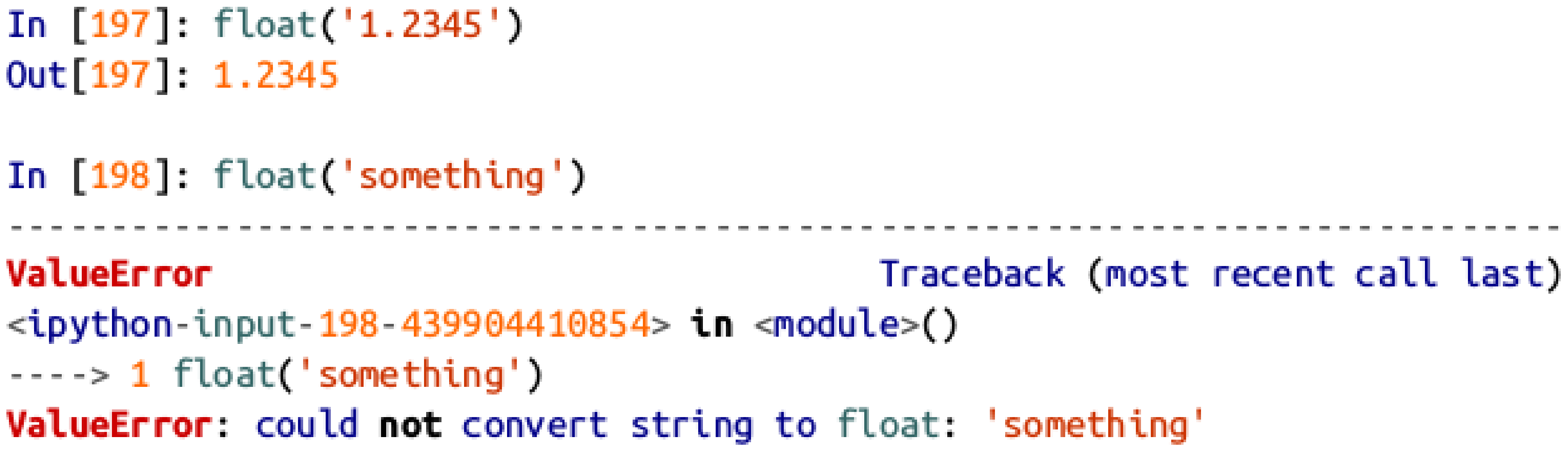
```
In [193]: import itertools

In [194]: first_letter = lambda x: x[0]

In [195]: names = ['Alan', 'Adam', 'Wes', 'Will', 'Albert', 'Steven']

In [196]: for letter, names in itertools.groupby(names, first_letter):
   .....:     print(letter, list(names)) # names is a generator
A ['Alan', 'Adam']
W ['Wes', 'Will']
A ['Albert']
S ['Steven']
```

# Python Basics : Some useful `itertools` functions



| Function | Description |
|---|---|
| `combinations(iterable, k)` | Generates a sequence of all possible k-tuples of elements in the iterable, ignoring order and without replacement (see also the companion function `combinations_with_replacement`) |
| `permutations(iterable, k)` | Generates a sequence of all possible k-tuples of elements in the iterable, respecting order |
| `groupby(iterable[, keyfunc])` | Generates `(key, sub-iterator)` for each unique key |
| `product(*iterables, repeat=1)` | Generates the Cartesian product of the input iterables as tuples, similar to a nested `for` loop |

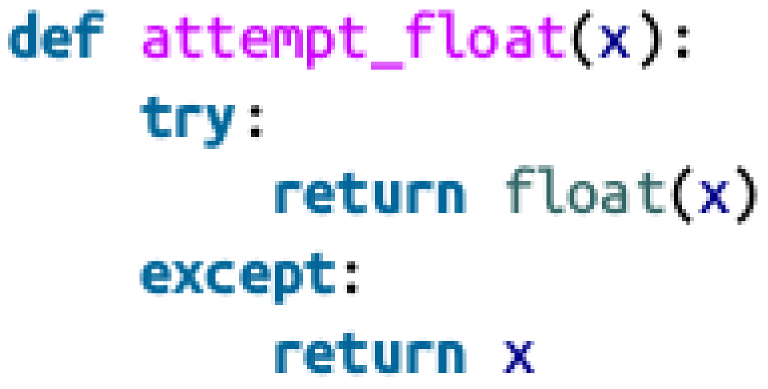
# Python Basics : Errors and Exception Handling

- Handling Python errors or `exceptions` gracefully is an important part of building robust programs.
- In data analysis applications, many functions only work on certain kinds of input.
- As an example, Python's `float` function is capable of casting a string to a floating-point number, but fails with `ValueError` on improper inputs
- 

```
In [197]: float('1.2345')
Out[197]: 1.2345

In [198]: float('something')
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-198-439904410854> in <module>()
----> 1 float('something')
ValueError: could not convert string to float: 'something'
```

# Python Basics : Errors and Exception Handling

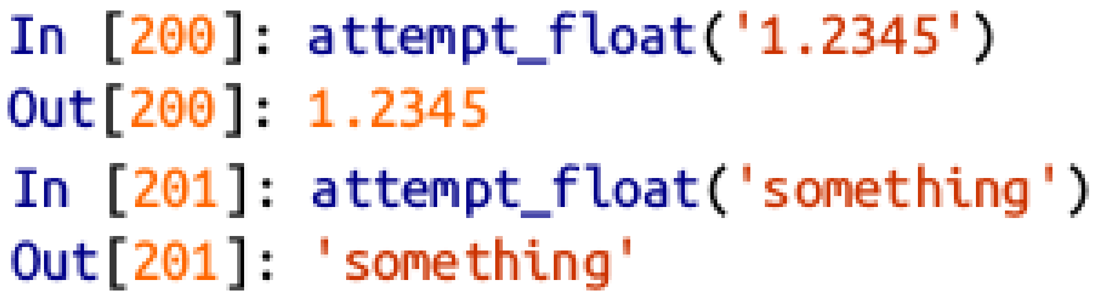
- Suppose we wanted a version of `float` that fails gracefully, returning the input argument.
- We can do this by writing a function that encloses the call to float in a `try / except` block



```
def attempt_float(x):
    try:
        return float(x)
    except:
        return x
```

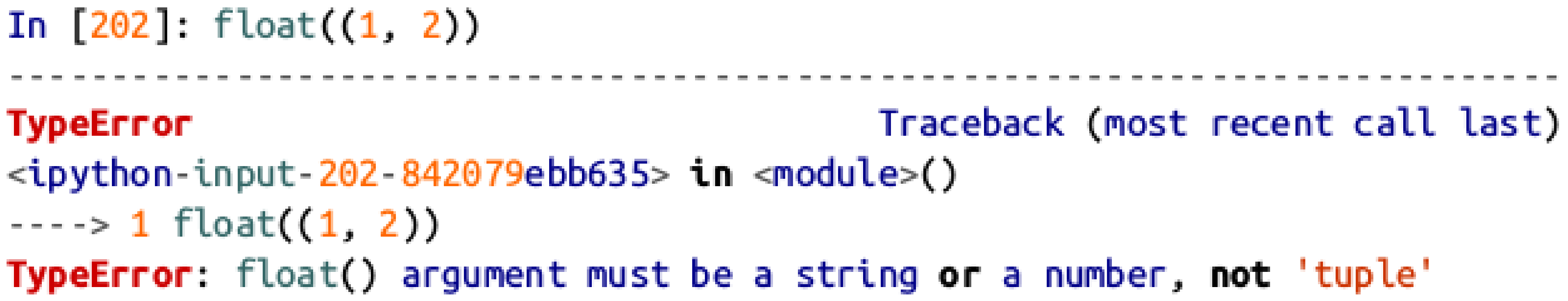
# Python Basics : Errors and Exception Handling

- The code in the except part of the block will only be executed if `float(x)` raises an exception.



```
In [200]: attempt_float('1.2345')
Out[200]: 1.2345
In [201]: attempt_float('something')
Out[201]: 'something'
```

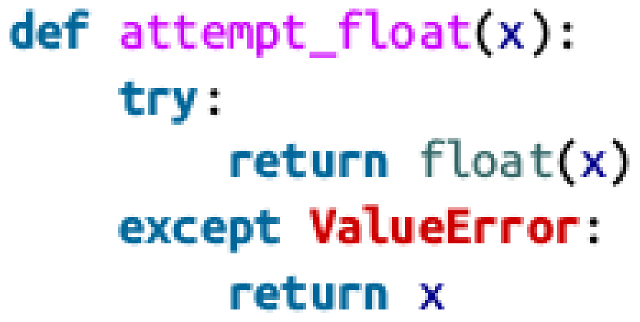
- You might notice that float can raise exceptions other than `ValueError`



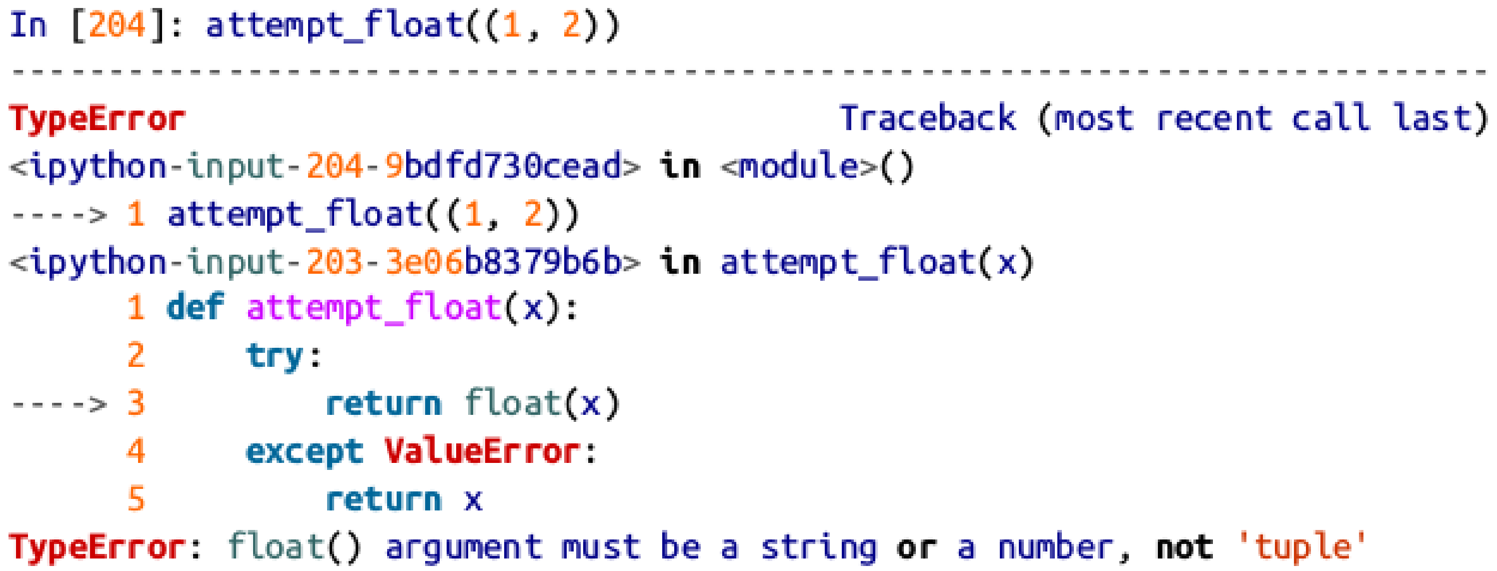
```
In [202]: float((1, 2))
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-202-842079ebb635> in <module>()
----> 1 float((1, 2))
TypeError: float() argument must be a string or a number, not 'tuple'
```

# Python Basics :
# Errors and Exception Handling

- You might want to only suppress ValueError , since a TypeError might indicate a legitimate bug in your program.



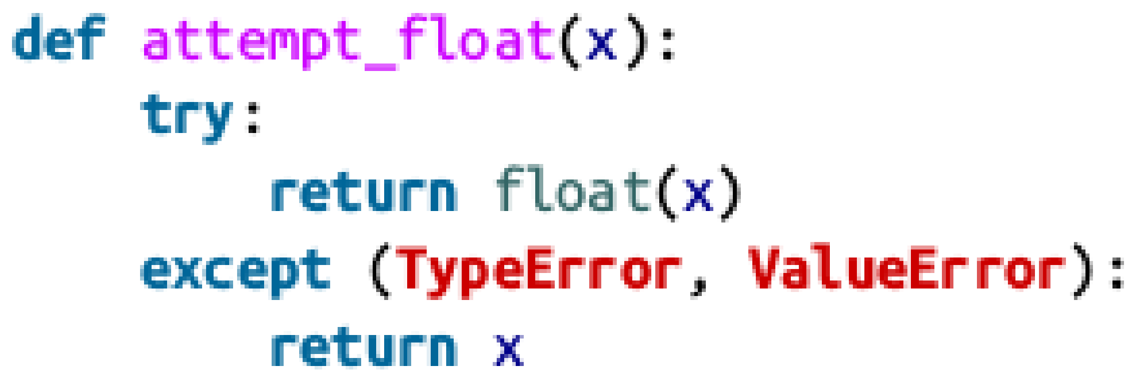
```python
def attempt_float(x):
    try:
        return float(x)
    except ValueError:
        return x
```



```
In [204]: attempt_float((1, 2))
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-204-9bdfd730cead> in <module>()
----> 1 attempt_float((1, 2))
<ipython-input-203-3e06b8379b6b> in attempt_float(x)
      1 def attempt_float(x):
      2     try:
----> 3         return float(x)
      4     except ValueError:
      5         return x
TypeError: float() argument must be a string or a number, not 'tuple'
```

# Python Basics : Errors and Exception Handling

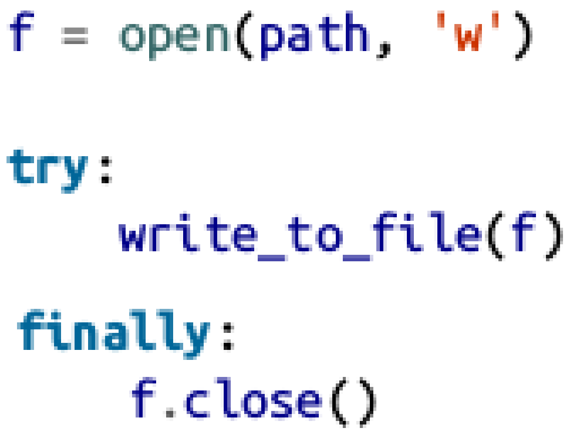
- You can catch multiple exception types by writing a tuple of exception types.



```python
def attempt_float(x):
    try:
        return float(x)
    except (TypeError, ValueError):
        return x
```

# Python Basics : Errors and Exception Handling

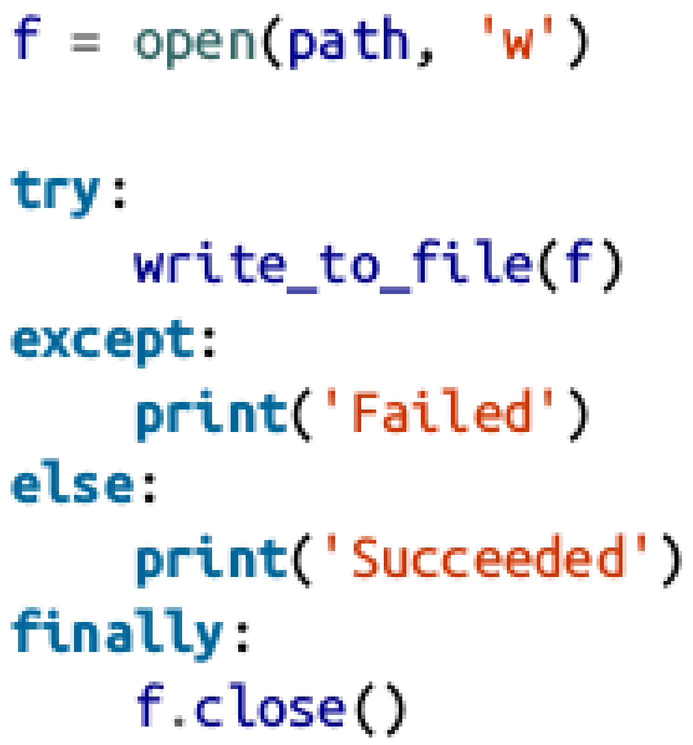
- In some cases, you may not want to suppress an exception, but you want some code to be executed regardless of whether the code in the try block succeeds or not.



```python
f = open(path, 'w')

try:
    write_to_file(f)
finally:
    f.close()
```

# Python Basics :
# Errors and Exception Handling

- You can have code that executes only if the `try:` block succeeds using `else`



```python
f = open(path, 'w')

try:
    write_to_file(f)
except:
    print('Failed')
else:
    print('Succeeded')
finally:
    f.close()
```

## Python Basics : Files and the Operating System

- Usually we use high-level tools like pandas.read_csv to read data files from disk into Python data structures.
- However, it’s important to understand the basics of how to work with files in Python.
- Fortunately, it’s very simple, which is one reason why Python is so popular for text and file munging.

# Python Basics : Files and the Operating System

- To open a file for reading or writing, use the built-in open function with a file path.

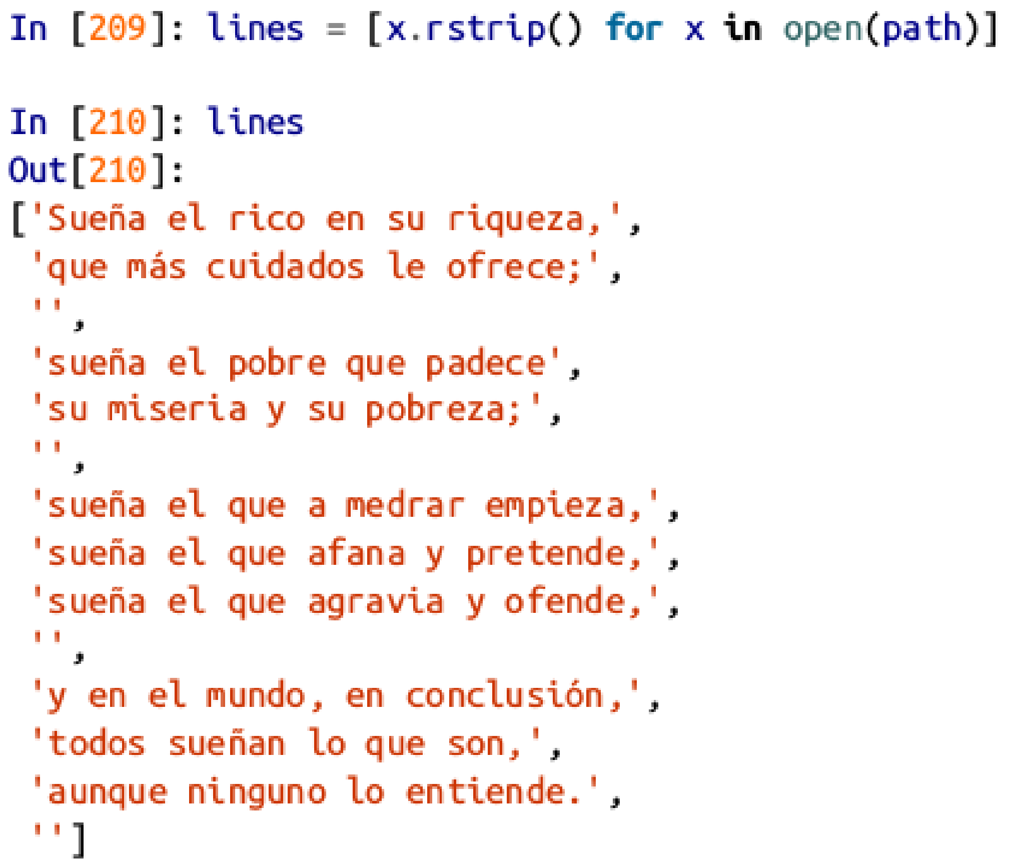
```
In [207]: path = 'examples/segismundo.txt'

In [208]: f = open(path)
```

- By default, the file is opened in read-only mode `'r'` . We can then treat the file handle `f` like a list and iterate over the lines like so :

```
for line in f:
    pass
```

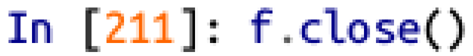
# Python Basics : Files and the Operating System

- The lines come out of the file with the `end-of-line (EOL)` markers intact, so you'll often see code to get an EOL-free list of lines in a file like



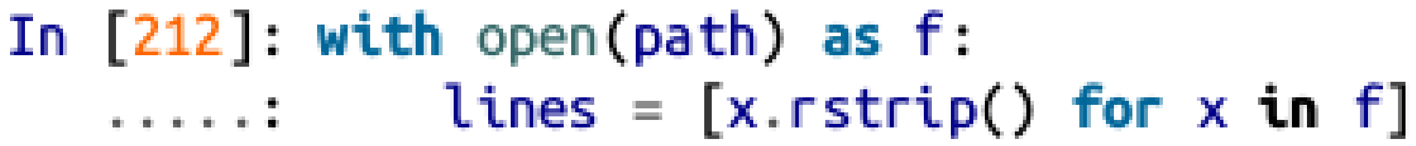
```
In [209]: lines = [x.rstrip() for x in open(path)]

In [210]: lines
Out[210]:
['Sueña el rico en su riqueza,',
 'que más cuidados le ofrece;',
 '',
 'sueña el pobre que padece',
 'su miseria y su pobreza;',
 '',
 'sueña el que a medrar empieza,',
 'sueña el que afana y pretende,',
 'sueña el que agravia y ofende,',
 '',
 'y en el mundo, en conclusión,',
 'todos sueñan lo que son,',
 'aunque ninguno lo entiende.',
 '']
```

# Python Basics : Files and the Operating System

- When you use open to create file objects, it is important to explicitly close the file when you are finished with it.
- Closing the file releases its resources back to the operating system: 

```
In [211]: f.close()
```

- One of the ways to make it easier to clean up open files is to use the with statement:



```
In [212]: with open(path) as f:
   .....:     lines = [x.rstrip() for x in f]
```

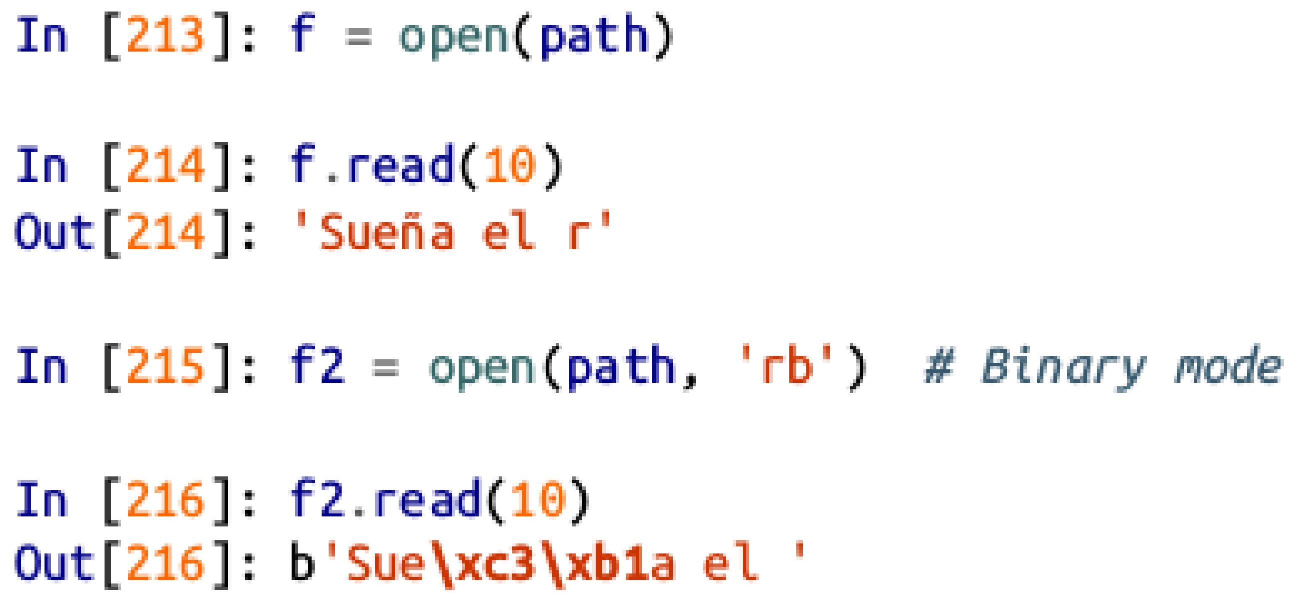
- This will automatically close the file f when exiting the with block.

## Python Basics : Files and the Operating System

- If we had typed `f = open(path, 'w')` , a new file at `examples/segismundo.txt` would have been created (be careful!), overwriting any one in its place.
- There is also the `'x'` file mode, which creates a writable file but fails if the file path already exists.

# Python Basics : Files and the Operating System

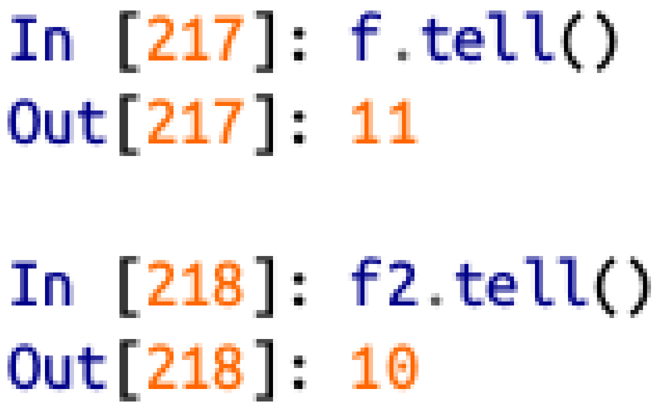
- For readable files, some of the most commonly used methods are `read` , `seek` , and `tell` .
- `read` returns a certain number of characters from the file.
- What constitutes a “character” is determined by the file’s encoding (e.g., UTF-8) or simply raw bytes if the file is opened in binary mode.
- .



```
In [213]: f = open(path)

In [214]: f.read(10)
Out[214]: 'Sueña el r'

In [215]: f2 = open(path, 'rb')  # Binary mode

In [216]: f2.read(10)
Out[216]: b'Sue\xc3\xb1a el '
```

# Python Basics : Files and the Operating System

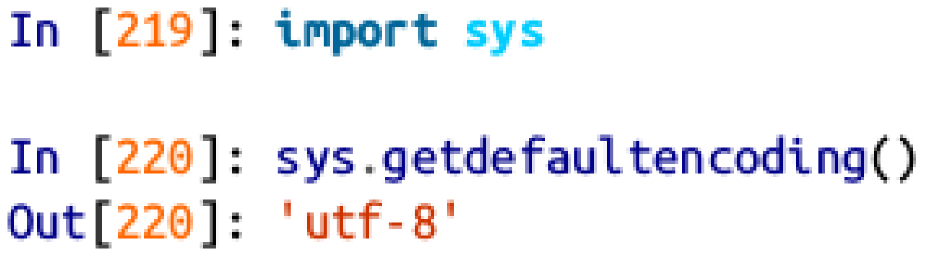
- The `read` method advances the file handle’s position by the number of bytes read. `tell` gives you the current position:



```
In [217]: f.tell()
Out[217]: 11

In [218]: f2.tell()
Out[218]: 10
```

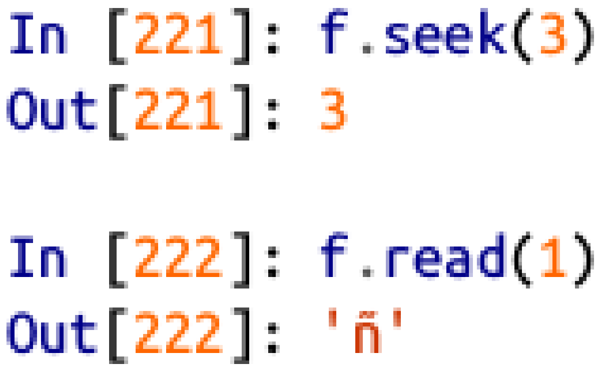
# Python Basics : Files and the Operating System

- You can check the default encoding in the sys module:



```
In [219]: import sys

In [220]: sys.getdefaultencoding()
Out[220]: 'utf-8'
```

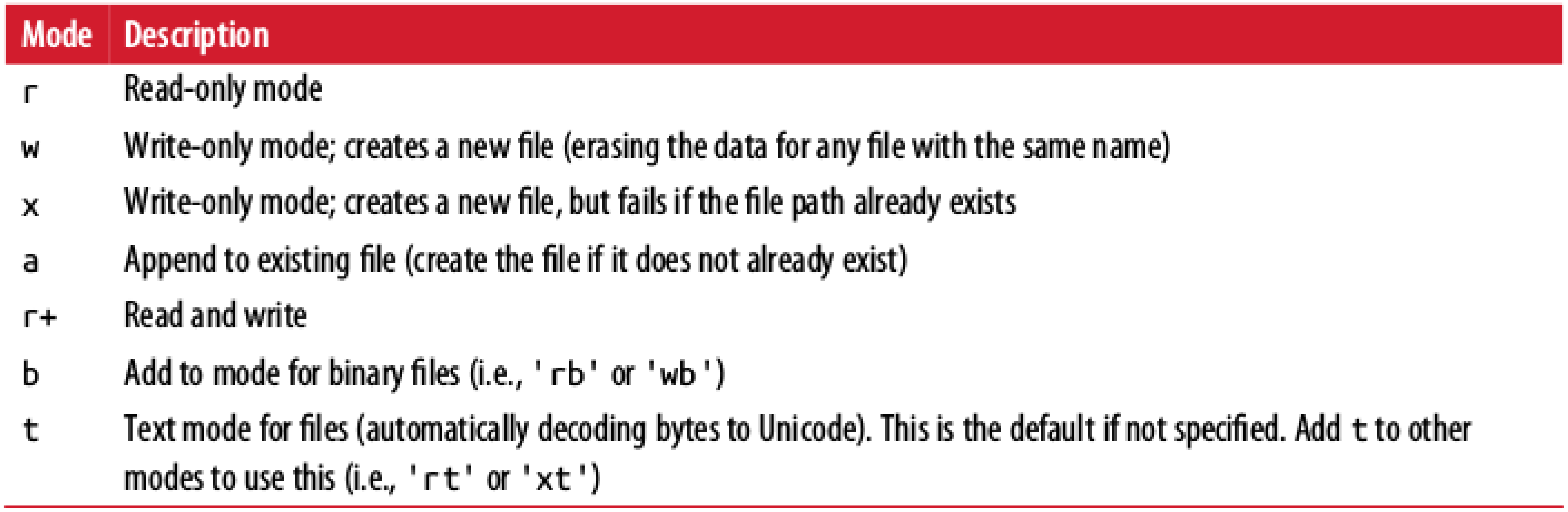
- `seek` changes the file position to the indicated byte in the file:



```
In [221]: f.seek(3)
Out[221]: 3

In [222]: f.read(1)
Out[222]: 'ñ'
```

# Python Basics :
# Files and the Operating System

- Python file modes table :



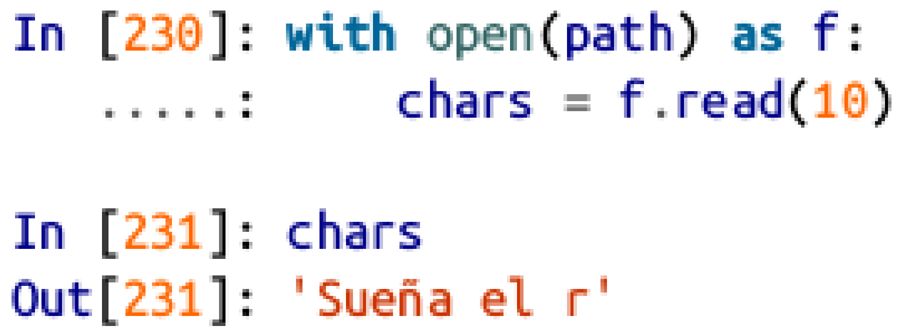
| Mode | Description |
|---|---|
| `r` | Read-only mode |
| `w` | Write-only mode; creates a new file (erasing the data for any file with the same name) |
| `x` | Write-only mode; creates a new file, but fails if the file path already exists |
| `a` | Append to existing file (create the file if it does not already exist) |
| `r+` | Read and write |
| `b` | Add to mode for binary files (i.e., `'rb'` or `'wb'`) |
| `t` | Text mode for files (automatically decoding bytes to Unicode). This is the default if not specified. Add `t` to other modes to use this (i.e., `'rt'` or `'xt'`) |

## Python Basics :
## Bytes and Unicode with Files

- The default behavior for Python files (whether readable or writable) is text mode, which means that you intend to work with Python strings (i.e., Unicode). This contrasts with binary mode, which you can obtain by appending b onto the file mode.

# Python Basics : Bytes and Unicode with Files

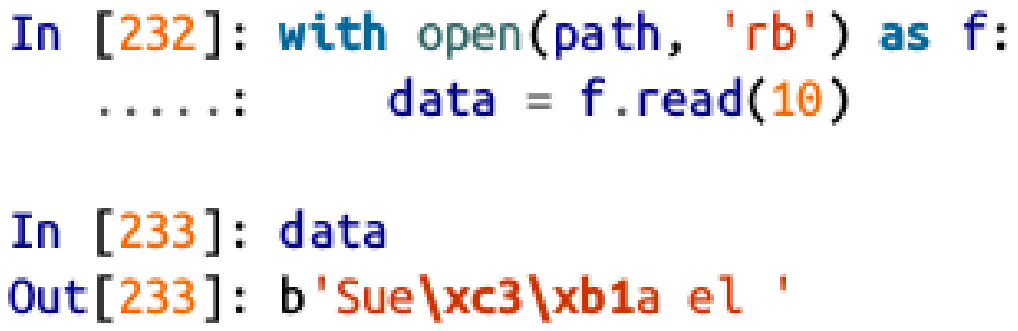
- Here we have a file which contains non-ASCII characters with UTF-8 encoding.



```
In [230]: with open(path) as f:
   .....:     chars = f.read(10)

In [231]: chars
Out[231]: 'Sueña el r'
```

# Python Basics : Bytes and Unicode with Files

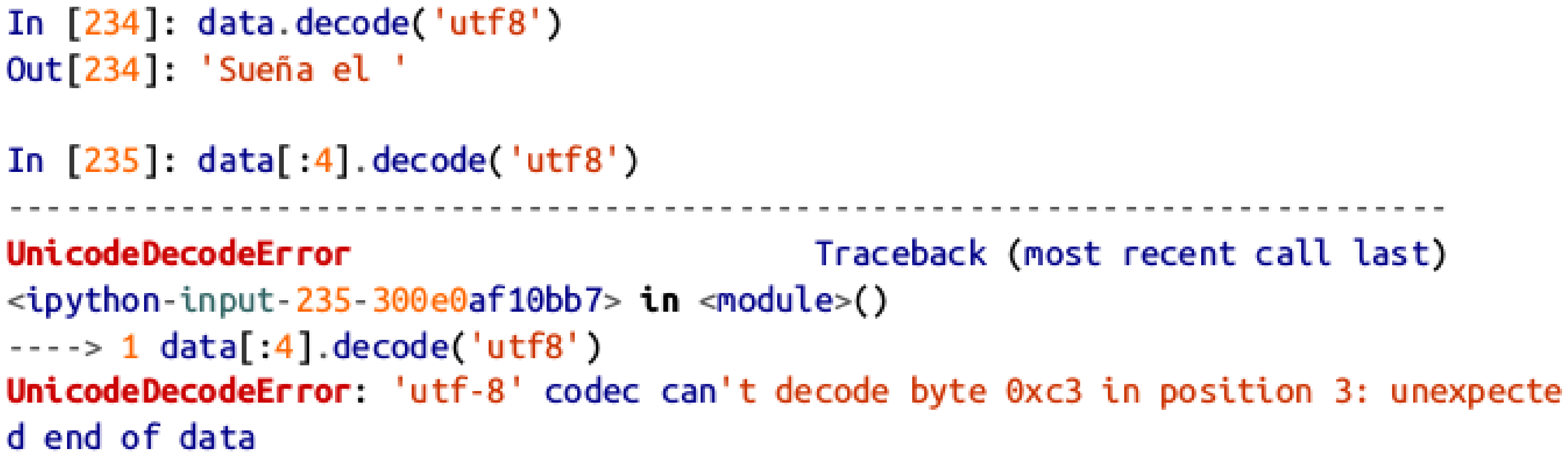
- UTF-8 is a variable-length Unicode encoding, so when I requested some number of characters from the file, Python reads enough bytes (which could be as few as 10 or as many as 40 bytes) from the file to decode that many characters.
- If I open the file in `'rb'` mode instead, read requests exact numbers of bytes:



```
In [232]: with open(path, 'rb') as f:
   .....:     data = f.read(10)

In [233]: data
Out[233]: b'Sue\xc3\xb1a el '
```

# Python Basics : Bytes and Unicode with Files

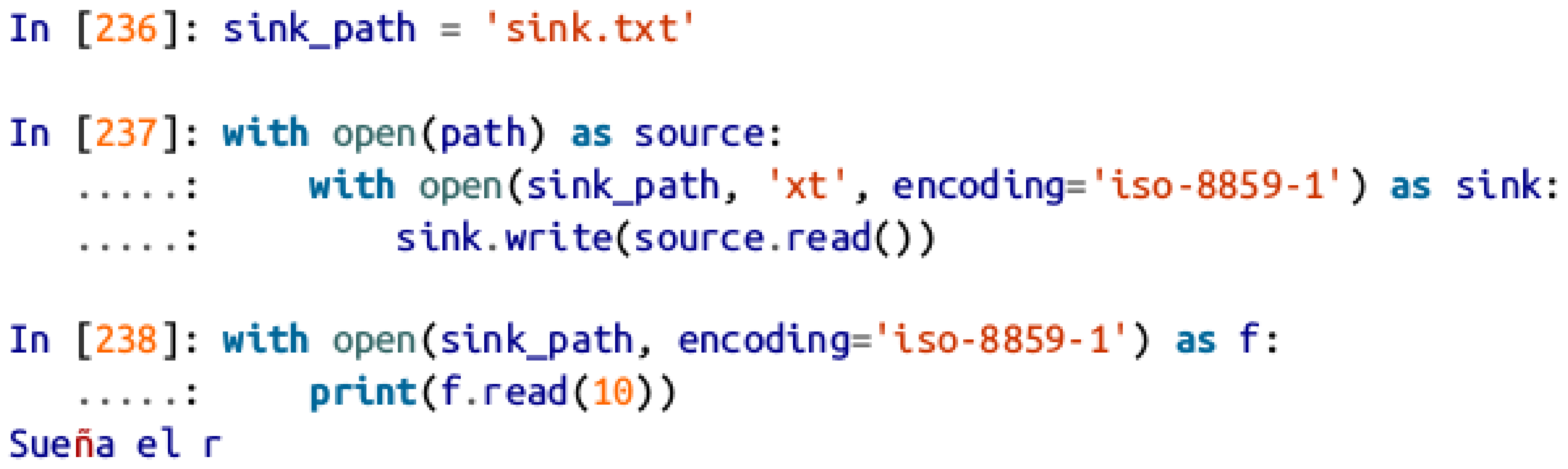
- Depending on the text encoding, you may be able to decode the bytes to a `str` object yourself, but only if each of the encoded Unicode characters is fully formed:



```
In [234]: data.decode('utf8')
Out[234]: 'Sueña el '

In [235]: data[:4].decode('utf8')
---------------------------------------------------------------------------
UnicodeDecodeError                        Traceback (most recent call last)
<ipython-input-235-300e0af10bb7> in <module>()
----> 1 data[:4].decode('utf8')
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xc3 in position 3: unexpecte
d end of data
```

# Python Basics : Bytes and Unicode with Files

- Text mode, combined with the encoding option of `open` , provides a convenient way to convert from one Unicode encoding to another:

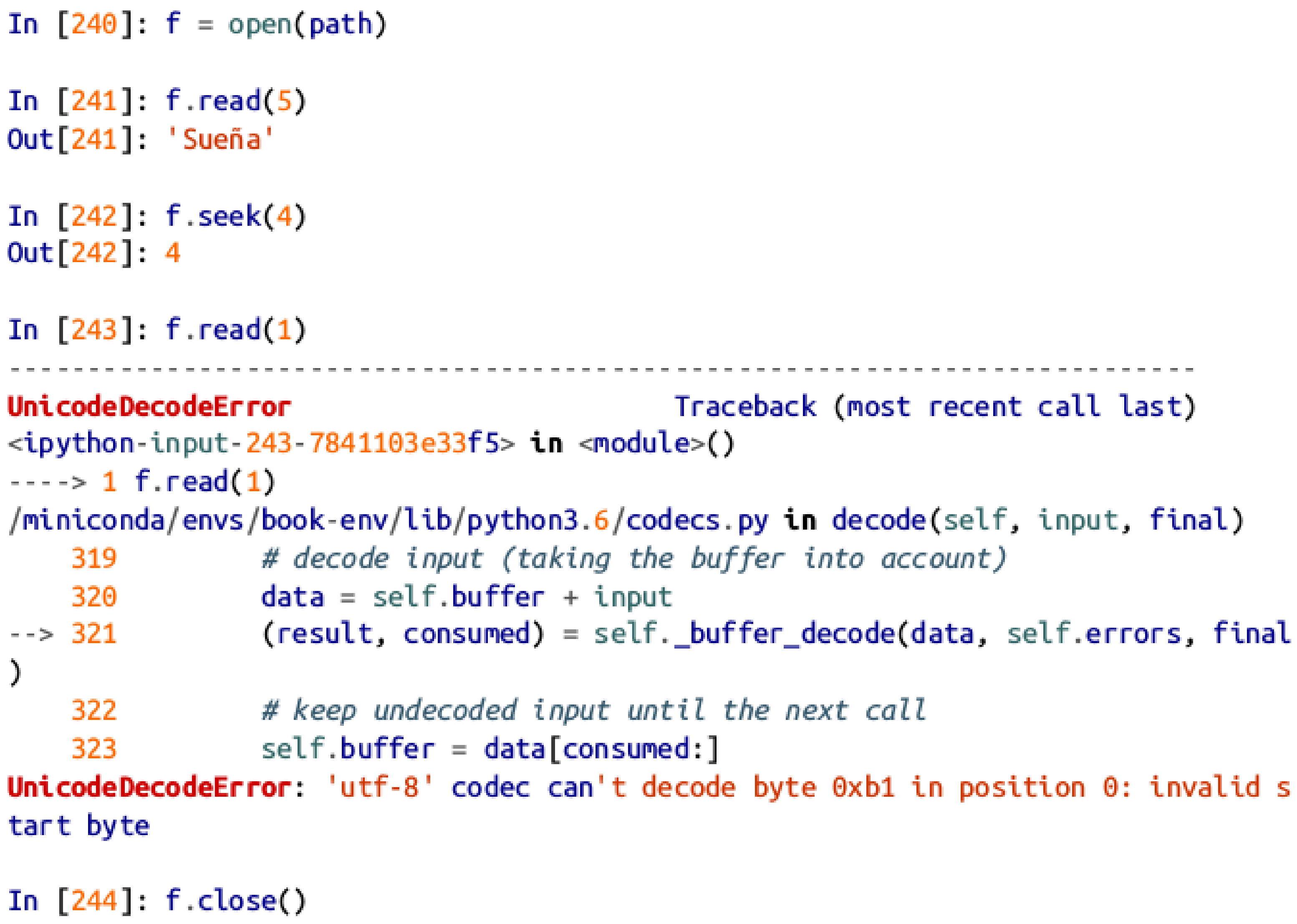


```
In [236]: sink_path = 'sink.txt'

In [237]: with open(path) as source:
   .....:     with open(sink_path, 'xt', encoding='iso-8859-1') as sink:
   .....:         sink.write(source.read())

In [238]: with open(sink_path, encoding='iso-8859-1') as f:
   .....:     print(f.read(10))
Sueña el r
```

## Python Basics :
## Bytes and Unicode with Files

- Beware using seek when opening files in any mode other than binary. If the file position falls in the middle of the bytes defining a Unicode character, then subsequent reads will result in an error.
- If you find yourself regularly doing data analysis on non-ASCII text data, mastering Python’s Unicode functionality will prove valuable.

# Python Basics : Bytes and Unicode with Files



```
In [240]: f = open(path)

In [241]: f.read(5)
Out[241]: 'Sueña'

In [242]: f.seek(4)
Out[242]: 4

In [243]: f.read(1)
---------------------------------------------------------------------------
UnicodeDecodeError                        Traceback (most recent call last)
<ipython-input-243-7841103e33f5> in <module>()
----> 1 f.read(1)
/miniconda/envs/book-env/lib/python3.6/codecs.py in decode(self, input, final)
    319         # decode input (taking the buffer into account)
    320         data = self.buffer + input
--> 321         (result, consumed) = self._buffer_decode(data, self.errors, final
)
    322         # keep undecoded input until the next call
    323         self.buffer = data[consumed:]
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xb1 in position 0: invalid s
tart byte

In [244]: f.close()
```
]