[Download notebook]

Context Managers

A *context manager* is a way to specify particular runtime contexts in your python code.

Most used example is file opening

In [2]:
```
with open("example.json") as file_handle:
    data = file_handle.read()
```

In the *context* of the `with` block there is an open file object `file_handle`. Since having a file open takes limited system resources, it's always important to close open files.

In [9]:
```
file_handle = open("example.json")
data = file_handle.read()
file_handle.close()
```

One should however always do

In [7]:
```
file_handle = open("example.json")
try:
    data = file_handle.read()
except IOError:
    pass
except Exception:
    print("Got exception")
else:
    print("Got no excpetion")
finally:
    file_handle.close()
```

In the case using the managed context, parts of setting up things, and ensuring that things get properly torn down, are handled automatically when *entering* and *exiting* the context.

This pattern is extremely common, therefore we have the `with` statement which uses what we call context managers to create contexts.

A context manager is a class whose objects (called a *context guard*) has an `__enter__` method and an `__exit__` method

In [13]:
```
class ContextManager(object):
    def __enter__(self):
        pass

    def __exit__(self, type, value, traceback):
        pass
```

When using the `with` statement what actually happens is (equivalent to)

In [20]:
```
context_guard = ContextManager()
value = context_guard.__enter__()
exc = True
try:
    try:
        var = value
        pass  # Here the code in the context
              # would go.
```

```
    except:
        exc = False
        if not context_guard.__exit__(*sys.exc_info()):
            raise

finally:
    if exc:
        context_guard.__exit__(None, None, None)
```

If `__enter__` returns a value, this can be caught by `... as var`

One can suppress exceptions under certain conditions with the return value of the `__exit__` method

Example:

In [1]:
```
import sys
from StringIO import StringIO

class redirect_stdout:
     def __init__(self, target):
         self.stdout = sys.stdout
         self.target = target

     def __enter__(self):
         sys.stdout = self.target

     def __exit__(self, type, value, tb):
         sys.stdout = self.stdout
```

In [2]:
```
out = StringIO()
```

In [3]:
```
with redirect_stdout(out):
    print("Prints to sys.stdout")
```

In [4]:
```
print("This too")
```

```
This too
```

In [5]:
```
out.getvalue()
```

Out[5]:
```
'Prints to sys.stdout\n'
```

Decorators

Say I have some functions defined in a module, among them these:

In [7]:
```
def my_add(x, y):
    return x + y

def my_subtract(x, y):
    return x - y

def my_multiply(x, y):
    return x * y
```

But I decide that whenever I'm calling this subset of functions, I want to print the parameters. That means I would have to modify each function this way:

In [8]:
```
def my_add(x, y):
    print(x, y)
    return x + y

my_add(6,7)
```

```
(6, 7)
```

Out[8]:
```
13
```

The printing functionality implementation would be the same for every function. This violates the principle of DRY: Don't Repeat Yourself.

A better way would be to make the implementation of the argument printing once, and apply that implementation to each function.

In [9]:
```
def print_args(function):
    def wrapper(*args, **kwargs):
        print("args: {}".format(args))
        print("kwargs: {}".format(kwargs))

        return function(*args, **kwargs)

    return wrapper
```

In [10]:
```
def my_add(x, y):
    return x + y

my_add = print_args(my_add)

my_add(3,4)
```

```
args: (3, 4)
kwargs: {}
```

Out[10]:
```
7
```

There is a python shorthand for the definition and modification, which is a **decorator**.

In [11]:
```
@print_args
def my_subtract(x, y):
    return x - y

my_subtract(5,2)
```

```
args: (5, 2)
kwargs: {}
```

Out[11]:
```
3
```

In [12]:
```
my_add(1,2,3,5, a1=1, a2=4)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call
last)
<ipython-input-12-17c0c25df71d> in <module>()
----> 1 my_add(1,2,3,5, a1=1, a2=4)

<ipython-input-9-a42d230fb22e> in wrapper(*args, **kwargs)
      4         print("kwargs: {}".format(kwargs))
```

```
      5
----> 6         return function(*args, **kwargs)
      7
      8     return wrapper

TypeError: my_add() takes exactly 2 arguments (6 given)
args: (1, 2, 3, 5)
kwargs: {'a1': 1, 'a2': 4}
```

There are some built-in decorators in python, and many packages uses them for functionality.

For example, in our production pipeline we use a `task` decorator from the package Celery to define available remotely executable "tasks"

(https://github.com/SciLifeLab/bcbb/blob/master/nextgen/bcbio/distributed/tasks.py#L84)

```
...

from celery.task import task
from bcbio.pipeline import lane

...

@task
def remove_contaminants(*args):
    return lane.remove_contaminants(*args)

...
```

`@property` decorator

You used to have a simple property of a class, but years later you realized this property would be more appropriate as returned value. But you don't want to break the functionality of the class for all the people who are already using your package.

In [13]:
```
class CLS(object):
    def __init__(self):
        self.a = 1
```

In [14]:
```
obj = CLS()
obj.a
```

Out[14]: 1

In [15]:
```
class CLS(object):
    def __init__(self):
        self.b = 3.
        self.c = 2.

    @property
    def a(self):
        return self.b / self.c
```

In [16]:
```
obj = CLS()

obj.a
```

Out[16]: 1.5

In [18]:
```
obj.b = 4.

obj.a
```

Out[18]:
```
2.0
```

Property objects have `getter`, `setter` and `deleter` attributes which can be used as decorators.

In [19]:
```
class CLS(object):
    def __init__(self):
        self.b = 3.
        self.c = 2.

    @property
    def a(self):
        return self.b / self.c

    @a.setter
    def a(self, value):
        self.c = 1.0
        self.b = value
```

In [20]:
```
obj = CLS()

print("Initial b, c, a: \n{}, {}, {}".format(obj.b, obj.c, obj.a))

obj.a = 5.0

print("")
print("b, c, a after a being set: \n{}, {}, {}".format(obj.b, obj.c,
obj.a))
```

```
Initial b, c, a:
3.0, 2.0, 1.5

b, c, a after a being set:
5.0, 1.0, 5.0
```

Class Decorators

One can also define decorators that take a class and return a new class. E.g. to add some pattern of methods or standard fields. This can be a quicker (to implement) alternative to MetaClasses.

Object Orientation

(Remember inheritance? See http://software-carpentry.org/4_0/oop/inherit.html if you need a refresher)

In [21]:
```
class A(object):
    a = []

class B(A):
    def __init__(self, name):
        self.name = name
        self.a.append(self.name)

    def print_list(self):
        print(self.a)
```

In [22]:
```
b1 = B("1")
b2 = B("2")
```

In [23]:
```
b1.print_list()
```

```
['1', '2']
```

In [25]:
```
b2.a is b1.a
```

Out[25]:
```
True
```

In [26]:
```
b3 = B("3")
b1.print_list()
```

```
['1', '2', '3']
```

In [27]:
```
A.a
```

Out[27]:
```
['1', '2', '3']
```

In [28]:
```
del b3
```

In [29]:
```
b1.print_list()
```

```
['1', '2', '3']
```

In [30]:
```
b3
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-30-c1b703401214> in <module>()
----> 1 b3

NameError: name 'b3' is not defined
```

In [31]:
```
class A(object):
    a = []


class B(A, object):
    def __init__(self, name):
        self.name = name
        self.a.append(self.name)

    def print_list(self):
        print(self.a)

    def __del__(self):
        self.a.remove(self.name)


b1 = B("1")
b2 = B("2")
b3 = B("3")
```

In [32]:
```
b1.print_list()
```

```
['1', '2', '3']
```

In [33]:
```
del b3
```

In [34]:
```
b1.print_list()
```

```
['1', '2']
```

Python has many method names starting and ending with __ which makes classes integrate more with the standard Python syntax.

These are normally called "Magic methods" or "Dunder methods".

This way one can make class objects behave in a nice way with any operator.

A complete list of these methods, with descriptions, is available at http://docs.python.org/2/reference/datamodel.html#special-method-names

A few examples are

```
__add__
__eq__
__and__
__str__
```

In [38]:
```
class A(object):
    a = []


class B(A):
    def __init__(self, name):
        self.name = name
        self.a.append(self.name)

    def print_list(self):
        print(self.a)

    def __del__(self):
        self.a.remove(self.name)

    def __add__(self, other):
        combined_name = "".join(sorted([self.name, other.name]))
        combined_b = B(combined_name)
        return combined_b

b1 = B("1")
b2 = B("2")
```

In [39]:
```
b12 = b2 + b1
```

In [37]:
```
b1.print_list()
```

```
['1', '2', '12']
```

It is usually a good design choice to make the `repr` of an object behave as the input of the constructur to create an equivalent object.

NumPy

NumPy is the heart of most calculations when using Python for scientific computing.

The most important part of it is that it provides a very efficient multidimensional array object

In [40]:
```
import numpy as np
```

The difference between an `array` and a `list` is that an `array` puts data in to contiguous memory blocks. While a `list` has a block of addresses referring to data.

This means `lists` are dynamic, you can store anything of any size in them, and grow them and shrink them however you want.

NumPy `arrays` can only have one datatype per instance. They *can* be grown and shrunk in size, but that is a (relatively) costly operation and should be avoided.

To instantiate an array one can pass a list to its constructor. But there are also efficient methods to create commonly needed arrays.

In [41]:
```
a = np.array([1., 2., 3.])
```

In [42]:
```
a
```

Out[42]:
```
array([ 1.,  2.,  3.])
```

Using the efficient array data structure, with numpy one can do *broadcasted* operations on the arrays

In [44]:
```
a * 4.0
```

Out[44]:
```
array([  4.,   8.,  12.])
```

In [45]:
```
type(a)
```

Out[45]:
```
numpy.ndarray
```

Arrays can have any number of dimensions, unlike a list which onle has one. (n-dimensional array)

Instantiated by lists of lists

In [46]:
```
b = np.array([[1, 4, 6], [5, 2, 2]])
```

In [47]:
```
b
```

Out[47]:
```
array([[1, 4, 6],
       [5, 2, 2]])
```

We can access items like on a list of lists

In [49]:
```
b[1][0]
```

Out[49]:
```
5
```

But arrays also have their own multidimensional accessors

In [50]:
```
b[1, 0]
```

Out[50]: 5

As well as multidimensional slicing

In [51]:
```
b[:, 1:2]
```

Out[51]:
```
array([[4],
       [2]])
```

Beware though that the ndarray slices are *views* in to the array rather than copies!

This means one can also broadcast on to what the view is referring to

In [58]:
```
b[1:2, 1:2] = 0
```

In [59]:
```
b
```

Out[59]:
```
array([[1, 0, 6],
       [5, 0, 2]])
```

In [60]:
```
b.shape
```

Out[60]:
```
(2, 3)
```

Good overview of most things one would need to know about arrays to use them efficiently:

- http://www.scipy.org/Tentative_NumPy_Tutorial

When operations are performed in a *vectorized* way, specialized C and Fortran methods will be used to perform the operation. Making them very quick and efficient.

Assignment

- Make a new file in your module called `session2.py`, and move the code from `__init__.py` in to it.
- Change your `getting_data.py` to import in this fashion: `from lastname.session2 import ...`
- Make a context manager which changes the current directory for the python script using it, and changes back to where you came from upon exit.

(Hint, look up `os.chdir` here http://docs.python.org/2/library/os.html)

Implement a class `CourseRepo` which takes a "surname" string in the constructor.

The class should have an attribute "required" which is a list of these strings:

- `".git"`
- `"setup.py"`
- `"README.md"`
- `"scripts/getting_data.py"`
- `"scripts/check_repo.py"`
- `"lastname/__init__.py"`
- `"lastname/session3.py"`

Where `"lastname"` is the "surname" string you gave to the constructor

`surname` should be a **property**, such that when it is set, the `required` attribute changes to reflect the new surname.

Example:

```
repo = CourseRepo("a")
print(repo.required[-1])
# prints a/session3.py

repo.surname = "b"
print(repo.required[-1])
# prints b/session3.py
```

The class should have a method `check()` which shall return `True` if all the strings in that list are existing files or directories.

(Hint: `os.path.exists`)

The context manager and class should be implemented in a file `session3.py` in your `lastname` module.

Then make a script, `scripts/check_repo.py`, which imports the context manager for changing current directory as well as the `CourseRepo` class from the module. Like this:

```
from lastname.session3 import CourseRepo, repo_dir
```

(where `repo_dir` is the name of the context manager)

This script should take an argument which is the absolute path to a repository.

(Hint: `sys.argv` or the built in `argparse` module)

The script should change directory to this given directory using the context manager. It should make an instance of `CourseRepo` using the final part of the absolute path, and call the `check()` method. If `check()` returns `True` the script shall print "PASS", otherwise the script should print "FAIL".

Example: If I call the script like

```
$ check_repo.py /Home/user/a
```

it should make a `CourseRepo` instance like in the example above (with `"a"`)

Like the other script, this script should also be installed when running `python setup.py install`.

Note: If your repo has a structure this script is not expecting, fixing it should be rather smooth using `git mv` for renaming/moving files and directories.