Science & Technology Facilities Council | UK Research and Innovation | NERC SCIENCE OF THE ENVIRONMENT

Python

Object-Oriented Programming (OOP)

Thanks to all contributors:

Alison Pamment, Sam Pepler, Ag Stephens, Stephen Pascoe, Kevin Marsh, Anabelle Guillory, Graham Parton, Esther Conway, Eduardo Damasio Da Costa, Wendy Garland, Alan Iwi, Matt Pritchard and Tommy Godfrey.

Centre for Environmental Data Analysis
SCIENCE AND TECHNOLOGY FACILITIES COUNCIL
NATURAL ENVIRONMENT RESEARCH COUNCIL

National Centre for Atmospheric Science
NATURAL ENVIRONMENT RESEARCH COUNCIL

National Centre for Earth Observation
NATURAL ENVIRONMENT RESEARCH COUNCIL

Computer science is the study of algorithms

Computer science is the study of algorithms

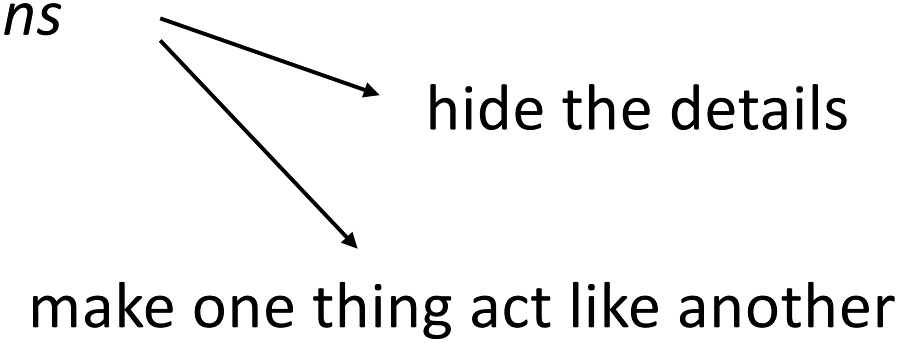
Computer *programming* is about creating and composing *abstractions*

Computer science is the study of algorithms

Computer *programming* is about creating and composing *abstractions* → hide the details

Computer science is the study of algorithms

Computer *programming* is about creating and composing *abstractions*

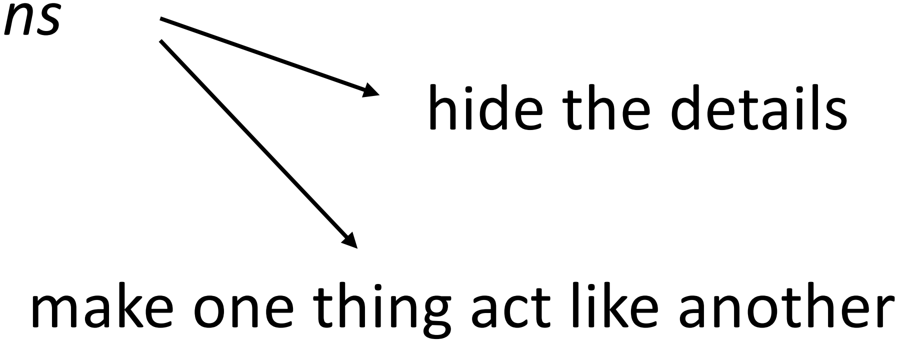


hide the details

make one thing act like another

Computer science is the study of algorithms

Computer *programming* is about creating and composing *abstractions*



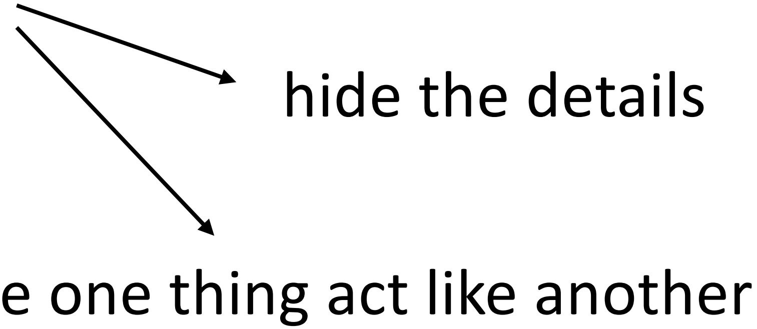
hide the details

make one thing act like another

Functions turn many steps into one (logical) step

Computer science is the study of algorithms

Computer *programming* is about creating and composing *abstractions*



hide the details

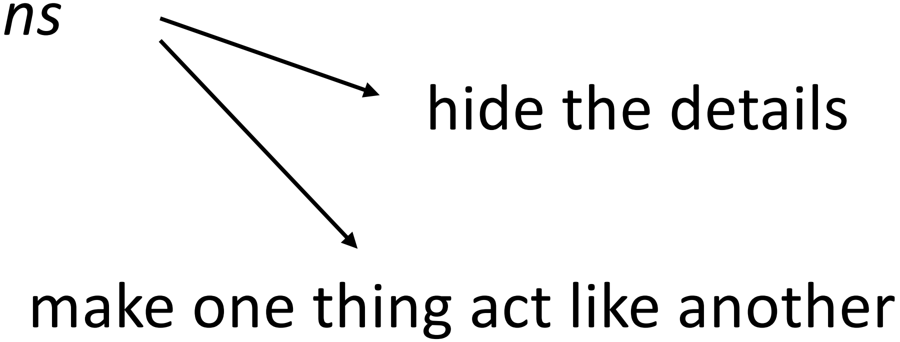
make one thing act like another

Functions turn many steps into one (logical) step

Libraries group functions to make them manageable

Computer science is the study of algorithms

Computer *programming* is about creating and composing *abstractions*



hide the details

make one thing act like another

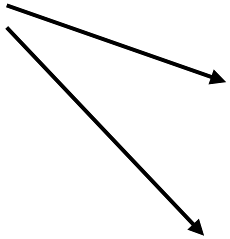
Functions turn many steps into one (logical) step

Libraries group functions to make them manageable

Classes and objects combine functions and data

Computer science is the study of algorithms

Computer *programming* is about creating and composing *abstractions*



hide the details

make one thing act like another

Functions turn many steps into one (logical) step

Libraries group functions to make them manageable

Classes and objects combine functions and data

And, if used properly, do much more as well

Let's see how OOP is useful in everyday Python:

```
>>> s = "some silly string"
>>> s.upper()
'SOME SILLY STRING'
>>> s.find("t")
12
>>> s.replace("silly", "sensible").title()
'Some Sensible String'
```

And you can actually interrogate this **object** *s* to find out their **methods**:

```
>>> dir(s)
['__add__', '__class__', '__contains__', '__delattr__', '__dir__',
'__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
'__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__iter__', '__le__', '__len__', '__lt__',
'__mod__', '__mul__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', 'capitalize', 'casefold',
'center', 'count', 'encode', 'endswith', 'expandtabs', 'find',
'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isascii',
'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric',
'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust',
'lower', 'lstrip', 'maketrans', 'partition', 'replace', 'rfind',
'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split',
'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate',
'upper', 'zfill']
```

And you can find out which **class** *s* is an **instance** of:

```
>>> type(s)
<class 'str'>
```

OOP Terminology (1)

class

Tell Python the definition of a new object.

object

Two meanings: the most basic type of thing, and any instance of a class.

instance

What you get when you tell Python to create a variable of given class.

def

How you define a method of a class.

self

Inside the methods in a class, self is a variable for the instance/object being accessed.

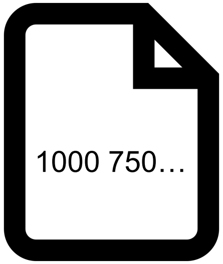
You can build your own **class** for your own domain:

```
class FileAnalyser():
    "A class above the rest"

    def __init__(self, path):
        items = open(path).read().split()
        self.data = []
        for item in items:
            self.data.append(float(item))

    def max(self):
        return max(self.data)

    def mean(self):
        return sum(self.data) / len(self.data)
```

Then create an **instance** of your **class** and use it:

```
$ cat some_data.txt
1000 750 500 250 0
```

Inside the data file...



some_data.txt

```
$ python

>>> from myclass import FileAnalyser
>>> da = FileAnalyser("some_data.txt")
>>> da.max()
1000.0
>>> da.mean()
500.0
```

You can make use of help() on your own class:

```
>>> help(FileAnalyser)
Help on class FileAnalyser in module myclass:

class FileAnalyser(builtins.object)
 |  FileAnalyser(path)
 |
 |  A class above the rest
 |
 |  Methods defined here:
 |
 |  __init__(self, path)
 |      Initialize self.  See help(type(self)) for accurate
signature.
 |
 |  max(self)
 |
 |  mean(self)
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

Let's look in detail at our class...:

```
class FileAnalyser():
    "A class above the rest"
```

Class Definition:
Defines the class name.

Optionally include a doc string below.

Let's look in detail at our class...:

```
class FileAnalyser():
    "A class above the rest"

    def __init__(self, path):
        items = open(path).read().split()
        self.data = []
        for item in items:
            self.data.append(float(item))
```

__init__ is the "constructor" method:

- Not necessary
- Very useful
- Always called when class is first created.

"self" means "belonging to this instance/object:

- Needed for all attributes that you want to be visible to every part of the object (shared).

Let's look in detail at our class...:

```
class FileAnalyser():
    "A class above the rest"

    def __init__(self, path):
        items = open(path).read().split()
        self.data = []
        for item in items:
            self.data.append(float(item))

    def max(self):
        return max(self.data)
```

Now we add more methods:

- "self" is always required as first argument.

Let's look in detail at our class...:

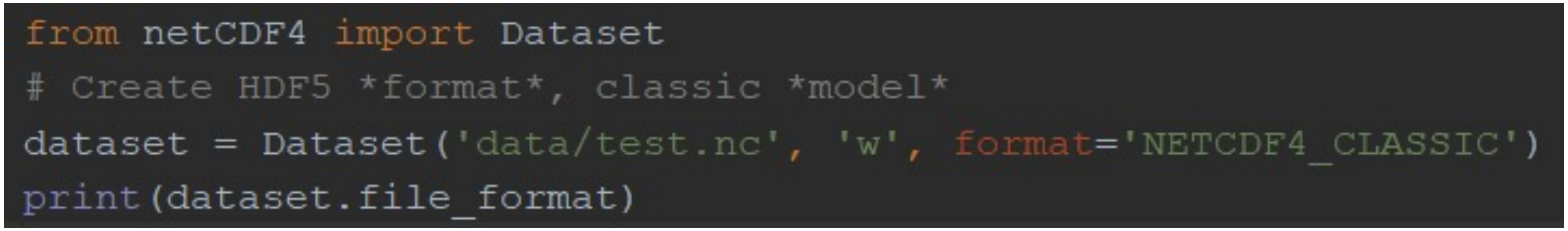
```
class FileAnalyser():
    "A class above the rest"

    def __init__(self, path):
        items = open(path).read().split()
        self.data = []
        for item in items:
            self.data.append(float(item))

    def max(self):
        return max(self.data)

    def mean(self):
        return sum(self.data) / len(self.data)
```

Examples of OOP

Most python packages use OOP extensively.

We'll come across many examples in the next sessions.

E.g.:



```
from netCDF4 import Dataset
# Create HDF5 *format*, classic *model*
dataset = Dataset('data/test.nc', 'w', format='NETCDF4_CLASSIC')
print(dataset.file_format)
```

A worked example

```
times = []
measurements = []

for i in range(1,32):
    date = f'2021-05-{i}'
    times, measurements = add_measurement(date, i, times, measurements)

# Print the data
print_measurements(times, measurements)
```

A worked example

```
times = []
measurements = []
```

Set up shared data containers

A worked example

```
times = []
measurements = []

for i in range(1,32):
    date = f'2021-05-{i}'
    times, measurements = add_measurement(date, i, times, measurements)
```

But also shared state

Pass in data to add

Re-assign shared state to take into account changes

A worked example

```
times = []
measurements = [

for i in range(0,31):
    date = f'2021-05-{i}'
    times, measurements = add_measurement(date, i, times, measurements)

# Print the data
print_measurements(times, measurements)
```

Pass in shared data

A worked example: Using classes

```
temp_store = DataStore()

for i in range(1,32):
    date = f'2021-05-{i}'
    temp_store.add_measurement(date,i)

# Print the temps
temp_store.print_measurements()
```

A worked example: Using classes

```
temp_store = DataStore()

for i in range(1,32):
    date = f'2021-05-{i}'
    temp_store.add_measurement(date,i)

# Print the temps
temp_store.print_measurements()
```

Create instance of DataStore
Shared data contained in class definition

A worked example: Using classes

```
temp_store = DataStore()

for                 2):
                    -05-{i}'
    temp_store.add_measurement(date,i)

# Print the temps
temp_store.print_measurements()
```

Create instance of DataStore
Shared data contained in class definition

No variable re-assignment

A worked example: Using classes

```
temp_store = DataStore()
```

Create instance of DataStore
Shared data contained in class definition

```
for                   2):
                      -05-{i}'
    temp_store.add_measurement(date,i)
```

No variable re-assignment

Only need to pass in things to add

```
# Print the temps
temp_store.print_measurements()
```

A worked example: Using classes

```
temp_store = DataStore()

for                 2):
                    -05-{i}'
    temp_store.add_measurement(date,i)

# Print the temps
temp_store.print_measurements()
```

Create instance of DataStore
Shared data contained in class definition

No variable re-assignment

Only need to pass in things to add

Don't need shared state

Number of things you need to remember are reduced