2 Overview

OOP is one of the major paradigms in programming.

The traditional programming paradigm (think Fortran, C, MATLAB, etc.) is called procedural.

It works as follows

- The program has a state corresponding to the values of its variables.
- Functions are called to act on these data.
- Data are passed back and forth via function calls.

In contrast, in the OOP paradigm

- data and functions are “bundled together” into “objects”

(FUNCTIONS in this context are referred to as methods)

2.1 Python and OOP

Python is a pragmatic language that blends object-oriented and procedural styles, rather than taking a purist approach.

However, at a foundational level, Python is object-oriented.

In particular, in Python, everything is an object.

In this lecture, we explain what that statement means and why it matters.
3 Objects

In Python, an object is a collection of data and instructions held in computer memory that consists of

1. a type
2. a unique identity
3. data (i.e., content)
4. methods

These concepts are defined and discussed sequentially below.

3.1 Type

Python provides for different types of objects, to accommodate different categories of data.

For example

```python
s = 'This is a string'
type(s)
```

str

```python
x = 42  # Now let's create an integer
type(x)
```

int

The type of an object matters for many expressions.

For example, the addition operator between two strings means concatenation

```python
'300' + 'cc'
```

'300cc'

On the other hand, between two numbers it means ordinary addition

```python
300 + 400
```

700

Consider the following expression

```python
'300' + 400
```

```
---------------------------------------------------------------------------
TypeError Traceback (most recent call last)
<ipython-input-5-263a89d2d982> in <module>
----> 1 '300' + 400

TypeError: can only concatenate str (not "int") to str
```

Here we are mixing types, and it’s unclear to Python whether the user wants to
• convert '300' to an integer and then add it to 400, or
• convert 400 to string and then concatenate it with '300'

Some languages might try to guess but Python is *strongly typed*

• Type is important, and implicit type conversion is rare.
• Python will respond instead by raising a *TypeError*.

To avoid the error, you need to clarify by changing the relevant type.

For example,

```
[6]: int('300') + 400  # To add as numbers, change the string to an integer
[6]: 700
```

### 3.2 Identity

In Python, each object has a unique identifier, which helps Python (and us) keep track of the object.

The identity of an object can be obtained via the `id()` function

```
[7]: y = 2.5
[7]: z = 2.5
[7]: id(y)
[7]: 148636839627136
[8]: id(z)
[8]: 148636839627088
```

In this example, `y` and `z` happen to have the same value (i.e., *2.5*), but they are not the same object.

The identity of an object is in fact just the address of the object in memory.

### 3.3 Object Content: Data and Attributes

If we set `x = 42` then we create an object of type `int` that contains the data *42*.

In fact, it contains more, as the following example shows

```
[9]: x = 42
[9]: x
[9]: 42
[10]: x.imag
[10]: 0
[11]: x.__class__
[11]: int
When Python creates this integer object, it stores with it various auxiliary information, such as the imaginary part, and the type.

Any name following a dot is called an attribute of the object to the left of the dot.

- e.g., \texttt{imag} and \texttt{__class__} are attributes of \texttt{x}.

We see from this example that objects have attributes that contain auxiliary information. They also have attributes that act like functions, called methods. These attributes are important, so let’s discuss them in-depth.

### 3.4 Methods

Methods are \textit{functions that are bundled with objects}.

Formally, methods are attributes of objects that are callable (i.e., can be called as functions)

```python
[12]: x = ['foo', 'bar']
callable(x.append)
[12]: True
[13]: callable(x.__doc__)
[13]: False
```

Methods typically act on the data contained in the object they belong to, or combine that data with other data

```python
[14]: x = ['a', 'b']
x.append('c')
s = 'This is a string'
s.upper()
[14]: 'THIS IS A STRING'
[15]: s.lower()
[15]: 'this is a string'
[16]: s.replace('This', 'That')
[16]: 'That is a string'
```

A great deal of Python functionality is organized around method calls.

For example, consider the following piece of code

```python
[17]: x = ['a', 'b']
x[0] = 'aa' # Item assignment using square bracket notation
x
[17]: ['aa', 'b']
```

It doesn’t look like there are any methods used here, but in fact the square bracket assignment notation is just a convenient interface to a method call.

What actually happens is that Python calls the \texttt{__setitem__} method, as follows

```python
[17]: ['aa', 'b']
```
In Python, *everything in memory is treated as an object*. This includes not just lists, strings, etc., but also less obvious things, such as

- functions (once they have been read into memory)
- modules (ditto)
- files opened for reading or writing
- integers, etc.

Consider, for example, functions.

When Python reads a function definition, it creates a *function object* and stores it in memory.

The following code illustrates

```python
[f(x): return x**2]  # 19
f
<function __main__.f(x)>  # 19
type(f)  # 20
function
id(f)  # 21
148636829724880
f.__name__  # 22
'f'
```

We can see that `f` has type, identity, attributes and so on—just like any other object.

It also has methods.

One example is the **call** method, which just evaluates the function

```python
f.__call__(3)  # 23
```

Another is the **dir** method, which returns a list of attributes.

Modules loaded into memory are also treated as objects.

4 Summary

In Python, *everything in memory is treated as an object*. This includes not just lists, strings, etc., but also less obvious things, such as

- functions (once they have been read into memory)
- modules (ditto)
- files opened for reading or writing
- integers, etc.

Consider, for example, functions.

When Python reads a function definition, it creates a *function object* and stores it in memory.

The following code illustrates

```python
[f(x): return x**2]  # 19
f
<function __main__.f(x)>  # 19
type(f)  # 20
function
id(f)  # 21
148636829724880
f.__name__  # 22
'f'
```

We can see that `f` has type, identity, attributes and so on—just like any other object.

It also has methods.

One example is the **call** method, which just evaluates the function

```python
f.__call__(3)  # 23
```

Another is the **dir** method, which returns a list of attributes.

Modules loaded into memory are also treated as objects.
This uniform treatment of data in Python (everything is an object) helps keep the language simple and consistent.

References