Objects

Every piece of data created and used by a Python program is called an *object*. There are simple objects such as numbers (both integers and floating point numbers) and Booleans (`True` and `False`), standard objects such as strings, lists, and tuples, and objects defined in special libraries or by the user, such as images, the canvas from the `itm421draw` module.

An objects stores some information—we call this the object's *state*. For instance, the state of a number object is the value of the number, the state of a string is the letters making up the string, and the state of a list is the length and the contents of the list.

In addition, objects provide *methods* to access the object's state, or to manipulate the state (that is, to change the information stored by the object). In Python, many methods are called using special syntax: for instance, `len(s)` really calls the method `s.__length__`, `a + b` calls the method `a.__add__(b)`, and `a in b` calls the method `b.__contains__(a)`.

Every object has a specific *type*. The type determines what you can do with the object, in particular, what methods it supports. For instance, you can write `a in b` when `b` is a list, but not when `b` is a number. Similar, the expression `a + b` means addition when `a` and `b` are numbers, but means concatenation when `a` and `b` are strings.

We have already used quite a few object types that are either built into the Python language, or that are provided by some library. In the future, we also want to create our own types. This is done using the `class` keyword. A *class* is a blueprint for objects—you create objects from the blueprint by just writing the class name.

Variables and the Heap

All objects are stored in a storage area of the Python interpreter called the *heap*.

A *variable* is really just a *name* for an object. So a variable does not "contain" a value, it is just a *reference* to an object. We will say that the variable *references* or *links to* this object.

The same object can have multiple names (that is, several variables "reference" the same object at the same time). The meaning of a name can change during the course of a program. This is really what an assignment instruction does: it makes the variable reference another object.

A name may also be unused. In that case we say that the variable has value `None`. You can test whether a variable `a` is unused by writing `a is None`, or using the negated form `a is not None`. (Note that this is a bit shorter and easier to read than the normal negated form `not (a is None)`.)

Mutable and immutable objects

Some objects cannot change their state once the object has been created. We call such objects *immutable*.

For instance, the number types are immutable: Once you create an object for the number `3`, for instance, this object will *always* store the number 3—it can never change its value. Students often find this confusing, because you can write something like this:

```
a = 3
a = 5
```

But note that in the second line we do not change the value of the object with value `3` created in the first line: Instead, we create a new object with value `5`, and change the variable `a` to link to the new object.

Strings are also immutable in Python: Once you have created a string object, you cannot change its value by replacing some letters or so. Any string method that seems to change the string actually returns a new string object. For instance

```
>>> s = "Hello ITM421"
>>> t = s.upper()
>>> s
'Hello ITM421'
```

Note that `s` remained unchanged by the call to the `upper()` method—it returns a new string object.

Tuples are another immutable type. Once you created a tuple, you cannot update its components.

Finally, there is a special immutable kind of set object called `frozenset`. We will see what it is good for later:

```
>>> s1 = set([1, 2, 3])
>>> s2 = frozenset([1, 2, 3])
>>> s1.add(9)
>>> s1
{1, 2, 3, 9}
>>> s2.add(9)
AttributeError: 'frozenset' object has no attribute 'add'
```

Most Python objects are *mutable*, meaning that their state can be changed after the object has been created. This can lead to "interesting" behavior such as the following:

```
>>> A = [1, 2, 3, 4]
>>> B = A
>>> A
[1, 2, 3, 4]
>>> B
[1, 2, 3, 4]
>>> A[2] = 99
>>> A
[1, 2, 99, 4]
>>> B
[1, 2, 99, 4]
```

We modified the contents of the list `A` — but then, when we look at the list `B`, we observe that its contents has also changed. What happened?

The answer, of course, is that there is *only one* list object. Both `A` and `B` are names for the same object, or, put differently, the two variables `A` and `B` reference the same list object.

We have to be careful when working with mutable objects that end up having multiple names. For this reason, immutable objects are generally preferred, except where efficiency is a problem. Today, with multi-core computers where several CPUs access the same data at the same time, immutable object designs become more and more important.

Local variables

As we discussed earlier, all objects are stored on the heap—no object can live anywhere else. Variables store just references to the objects inside the heap, they do not actually contain a value.

So where are these references stored? Some references are inside objects. Consider, for instance, this example:

```
a = [ "Hello ITM412", 13 ]
```

Here `a` is the name for a list object. *Inside* the list object are references to two other objects, a string and an integer.

But what about the variable names used in your program? Consider this little function:

```
def test(m):
  k = m + 27
  s = "Hello World"
  A = [ len(s), k, m ]
```

This function has *four local variables*: `m`, `k`, `s`, and `A`. Yes, you read that correctly: the parameter `m` is also a local variable. Parameters are different from other local variables only through the fact that they are given a value from arguments when the function is called.

Every time the function `test` is called, the Python interpreter creates a special memory area called an *activation record* (because the function is being activated) or *stack frame* (we'll get back to this name later). All local variables are stored inside the stack frame. When the function returns, the stack frame is destroyed, and the local variables no longer exist.

Now that we have covered local variables, you may wonder where *global variables* are stored. There actually is an object for every Python module of your program, and this object stores the global variables (so, in a sense, global variables are like fields in an object).

Garbage collection

A typical program creates many small objects that are only used for a brief time, and not needed afterwards. This results in the heap filling up with unused objects, and at some point the heap is simply full and cannot store any new objects.

At this point, the Python interpreter needs to perform *garbage collection*. It inspects each and every object on the heap and determines whether the object is *garbage*—that is, the object can no longer be used. Those garbage objects are discarded, and the program can continue.

How can the interpreter know whether an object is still useful? Well, to be useful, an object needs to have a name: there needs to be a variable (or field in another object) that references it. So the Python interpreter starts from all the global variables and all the active local variables, and follows those references to find every object that is still reachable. All objects not found during this process are garbage.

If you have programmed in `C` or `C++` before, you have learnt that for every object that you allocate (with `malloc` or `new`), you have to free the memory later (with `free` or `delete`). `C` and `C++` do not have garbage collection, so the programmer is responsible for managing the heap. The result is that `C` and `C++` programs often have *memory leaks*: Some objects are created, but never destroyed, so the heap slowly fills up with unused objects, and the program uses more and more of your computer's memory. In Python, Java, Kotlin, and other languages with garbage collection we do not have to worry about memory leaks.

Modern garbage collectors are more sophisticated than the scheme described above—they may store some bookkeeping information with each object to figure out quickly when it becomes garbage (and then discard it immediately). They may also perform some garbage collection incrementally, in parallel with the computation. Efficient garbage collection is an active research area.

Defining classes

We will study how to define a class using the `Token` class used for calculator implementation. Here is the class definition:

```
class Token(object):
  def __init__(self, text, pos, type):
    self.pos = pos
    self.type = type
    if type == "number":
      self.value = float(text)
    else:
      self.value = text
```

The keyword `class` is followed by the name of the class, by convention starting with a capital letter. In parenthesis follows the name of the "superclass"—in this course, we will nearly always write `object` here to indicate that there is no specific superclass. You'll learn about superclasses, subclasses, and class hierarchies when you study object-oriented programming.

The keyword `def` *inside* a class starts a *method definition*. Here, we have one method with the special, magic name `__init__`. This method is the *constructor* of the `Token` class—it is executed every time an object of type `Token` is created. It's duty is to initialize the state of the object correctly.

Our token objects are very simple: they just store the type of the token (as a string, namely one of the values "number", "identifier", "symbol", or "stop"), the token's value (either a numeric value for number tokens, or a string for identifier and symbol tokens), and finally the position in the input string where the token starts (this is only used for reporting errors).

The constructor takes three parameters for these pieces of information, and stores them in the *fields* `type`, `value`, and `pos` of the `Token` object. But note that the constructor method actually has *four* parameters. The first parameter, which is always called `self`, refers to the `Token` object that is being created. Therefore assigning to `self.type` creates the `type` field inside the object, and similar for `self.value` and `self.pos`.

With this code we can start creating `Token` objects, like this:

```
>>> t = Token("3.5", 7, "number")
>>> t.type
'number'
>>> t.value
3.5
>>> h = Token("*", 12, "symbol")
>>> h.type
'symbol'
>>> h.value
'*'
```

Note that once the object has been set up by the constructor, we can access the fields containing its state.

Defining methods

Even though `Token` is a simple class, it will be useful to give it a few methods to make our parsing code more readable. For instance, instead of `t.type == "number"`, we want to write `t.isNumber()`, and instead of

```
t.type == "symbol" and t.value == "*"
```

we want to write `t.isSymbol("*")`. Here are the method definitions. Again they go inside the class definition:

```
def isNumber(self):
  return self.type == "number"

def isSymbol(self, s):
  return self.type == "symbol" and self.value == s

def isIdentifier(self):
  return self.type == "identifier"

def isStop(self):
  return self.type == "stop"
```

Note that the first parameter of each method is again called `self`. When calling the method, this parameter will reference the `Token` object itself. For instance, when we call `t.isSymbol("*")`, then `self` will be the object `t`, and `s` will be the string `"*"`.

Converting to strings

It is often useful to add methods to a class that will print the object in a nice format. Again there is a magic method name, namely `__str__`. This method is called whenever objects are converted to strings with the `str` function. This happens, for instance, when printing an object using the `print` function: it automatically converts its argument to a string.

Here is the definition for the `__str__` method of our `Token` class:

```
def __str__(self):
  if self.isNumber():
    return "Number: %g" % self.value
  if self.isIdentifier():
    return "Identifier: %s" % self.value
  if self.isStop():
    return "Stop"
  return "Symbol: %s" % self.value
```

Again, `self` refers to the `Token` object. Here are some tests:

```
>>> str(h)
'Symbol: *'
>>> str(t)
'Number: 3.5'
>>> print(h)
Symbol: *
>>> print(t)
Number: 3.5
```

Note how `print` first converts the `Token` object to a string.

Python actually supports a *second form* of string conversion, using the magic method name `__repr__`. This method is called using the `repr` function. It is also used by the interactive mode to display the result of each line of input.

The `__repr__` method is meant for debugging, it is used by the programmer for testing and in the interactive mode, not in code meant for the "end user." Therefore, the result often looks quite different.

```
>>> print(7)
7
>>> print(repr(7))
7
>>> print("hello")
hello
>>> print(repr("hello"))
'hello'
```

For numbers, there is no difference between the result of `str` and `repr`. For strings, however, we note that the output of `repr` contains an extra pair of quotes. The difference becomes even more pronounced if we include some special characters in the string (`\t` means a tab character, `\n` is a newline character):

```
>>> s = "Hello\tITM412!\n"
>>> print(s)
Hello   ITM412!

>>> print(repr(s))
'Hello\tITM412!\n'
```

When printing the string, the special characters are interpreted. The output of `repr`, however, contains special characters. The effect is that we could take the output of `repr` and feed it back into the Python interpreter to create a copy of the same string.

This is a general design principle for the `repr` conversion: its output often has a format that can be copied back into the interpreter to create a copy of the object. Let's use this design for our `Token` class:

```
  def __repr__(self):
    return "Token(%s, %d, %s)" % (repr(self.value), self.pos, repr(self.type))
```

Note that this works correctly both for number and string values:

```
>>> print(repr(t))
Token(3.5, 7, 'number')
>>> print(repr(h))
Token('*', 12, 'symbol')
```

You can copy this output back into the interpreter to recreate the `Token` object.

One good reason to define a `repr` conversion is that it allows you to look at lists of your objects:

```
>>> toks = tokenize("a23 * (3 - 4 * x)")
>>> toks
[Token('a23', 0, 'identifier'), Token('*', 4, 'symbol'),
  Token('(', 6, 'symbol'), Token(3.0, 7, 'number'), Token('-', 9, 'symbol'),
  Token(4.0, 11, 'number'), Token('*', 13, 'symbol'),
  Token('x', 15, 'identifier'), Token(')', 16, 'symbol'),
  Token('', 17, 'stop')]
```

This works because the string conversion for list objects uses the `repr` conversion for each list element.

Object implementation and client code

Ideally, code that *uses* an object of a certain type should not need to know how that object is implemented. For instance, we can use Python's list objects without needing to know how the Python interpreter implements them so efficiently. (But we'll figure it out, I promise!)

It therefore often makes sense to separate the code that *implements* a class from the code that *uses* the class. We usually have one file that contains the definitions for one class (or perhaps a few closely related classes), and perhaps some functions that work with objects of this type. Such a file is called a Python *module*.

The code that *uses* the object is called *client code*. Think about your class providing some kind of service, the client code is a "customer" for this service—hence the name "client code".

To use your class, client code must *import* the module defining the class. In our case, we need a line

```
import tokens
```

We can then use the `Token` class and the `tokenize` function by prefixing them with the module name:

```
>>> import tokens
>>> h = tokens.Token("3.5", 7, "number")
>>> toks = tokens.tokenize("3 * 5 - 7 / x")
```

Sometimes it's annoying having to write the module name all the time. In that case we can import all the classes and functions from the module into the client code's *name space*, like this:

```
>>> from tokens import *
>>> h = Token("3.5", 7, "number")
>>> toks = tokenize("3 * 5 - 7 / x")
```

Object equality

The `==` operator can be used to test equality of any kind of object. However, by default it is not very useful:

```
>>> h
Token(3.5, 7, 'number')
>>> h1 = h
>>> h2 = Token("3.5", 7, "number")
>>> h2
Token(3.5, 7, 'number')
>>> h == h1
True
>>> h == h2
False
```

Note that `h == h1` is true, because `h` and `h1` are actually the same object. However, `h == h2` returns false, even though the two objects have exactly the same state.

We can fix this by adding a method with another magic name to our `Token` class:

```
def __eq__(self, rhs):
  return (self.type == rhs.type and
          self.value == rhs.value)
```

Note that in this example we do not check the `pos` field, so two `Token` objects are considered equal when type and value are identical.