User-defined Classes

Our calculator works with tokens, so we define a `Token` type.

A `Token` has several attributes:

- A type (number, identifier, symbol, or stop);
- A value (number or string);
- A position in the input string (for error reporting).

Attributes are stored in the fields of the object.

```
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
```


Constructors

Objects can have a special method `__init__`, called a constructor. Whenever an object of this type is created, the constructor is called.

```
>>> t = Token("3.5", 7, "number")
>>> s = Token("abc", 9, "identifier")
>>> h = Token("*", 12, "symbol")
```

We can look at the fields of the objects:

```
>>> h.value
'*'
>>> h.type
'symbol'
>>> t.type
'number'
>>> t.value
3.5
>>> s.value
'abc'
>>> s.type
'identifier'
```

Note that the first parameter `self` of the method refers to the object itself. No argument is given in the constructor call.

Methods

We add a few methods to the class:

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

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

Each method needs the `self` parameter!

Using the methods:

```
>>> t.isNumber()
True
>>> h.isNumber()
False
>>> h.isSymbol("*")
True
```


String conversion

We can make conversion to strings even nicer: `str(tok)` calls the special method `__str__`:

```
  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
```

```
>>> str(h)
'Symbol: *'
>>> print(h)
Symbol: *
```

`print` automatically converts its arguments to `str`

"Representation"

Python supports a second kind of string conversion, often used for debugging. It uses the special method `__repr__`.

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

Often the `repr` form can be pasted into the interpreter to create the same object.

```
>>> s
Token('abc', 9, 'identifier')
>>> print(s)
Identifier: abc
```

```
>>> a = "Hello\n"
>>> print(a)
Hello

>>> a
'Hello\n'
```


Equality

The comparison operators ==, !=, < etc. do not work automatically for objects:

```
>>> Token("abc", 2, "identifier") ==  \
      Token("abc", 2, "identifier")
False
```

We can define equality through the special method `__eq__`:

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

There are special methods for other operators (addition, multiplication, indexing, length, etc.) as well.

Client code

Client code is code that uses an object.

The calculator is written by making use of the `Token` type. This type allows us to express the parser naturally.

The calculator does not need to know how `Token` is implemented.

It is natural to keep the client code in a separate file from the code that defines and implements the `Token` class.

```
import tokens
```