Mark Summerfield

Foreword by Doug Hellmann,
Senior Developer, DreamHost

Python in Practice

Create Better Programs Using Concurrency, Libraries, and Patterns

Developer's Library

Python in Practice

```
    def add(self, command):
        if not isinstance(command, Command):
            raise TypeError("Expected object of type Command, got {}".
                    format(type(command).__name__))
        self.__commands.append(command)

    def __call__(self):
        for command in self.__commands:
            command()

    do = __call__

    def undo(self):
        for command in reversed(self.__commands):
            command.undo()
```

The `Command.Macro` class is used to encapsulate a sequence of commands that should all be done—or undone—as a single operation.* The `Command.Macro` offers the same interface as `Command.Commands`: `do()` and `undo()` methods, and the ability to be called directly. In addition, macros provide an `add()` method through which `Command.Commands` can be added.

For macros, commands must be undone in reverse order. For example, suppose we created a macro and added the commands A, B, and C. When we executed the macro (i.e., called it or called its `do()` method), it would execute A, then B, and then C. So when we call `undo()`, we must execute the `undo()` methods for C, then B, and then A.

In Python, functions, bound methods, and other callables are first-class objects that can be passed around and stored in data structures such as `lists` and `dicts`. This makes Python an ideal language for implementations of the Command Pattern. And the pattern itself can be used to great effect, as we have seen here, in providing do–undo functionality, as well as being able to support macros and deferred execution.

3.3. Interpreter Pattern

The Interpreter Pattern formalizes two common requirements: providing some means by which users can enter nonstring values into applications, and allowing users to program applications.

At the most basic level, an application will receive strings from the user—or from other programs—that must be interpreted (and perhaps executed) appropriately. Suppose, for example, we receive a string from the user that is supposed

* Although we speak of macros executing in a single operation, this operation is not atomic from a concurrency point of view, although it could be made atomic if we used appropriate locks.

to represent an integer. An easy—and unwise—way to get the integer's value is like this: `i = eval(userCount)`. This is dangerous, because although we hope the string is something innocent like `"1234"`, it could be `"os.system('rmdir /s /q C:\\\\')"`.

In general, if we are given a string that is supposed to represent the value of a specific data type, we can use Python to obtain the value directly and safely.

```
try:
    count = int(userCount)
    when = datetime.datetime.strptime(userDate, "%Y/%m/%d").date()
except ValueError as err:
    print(err)
```

In this snippet, we get Python to safely try to parse two strings, one into an `int` and the other into a `datetime.date`.

Sometimes, of course, we need to go beyond interpreting single strings into values. For example, we might want to provide an application with a calculator facility or allow users to create their own code snippets to be applied to application data. One popular approach to these kinds of requirements is to create a DSL (Domain Specific Language). Such languages can be created with Python out of the box—for example, by writing a recursive descent parser. However, it is much simpler to use a third-party parsing library such as PLY (`www.dabeaz.com/ply`), PyParsing (`pyparsing.wikispaces.com`), or one of the many other libraries that are available.*

If we are in an environment where we can trust our applications' users, we can give them access to the Python interpreter itself. The IDLE IDE (Integrated Development Environment) that is included with Python does exactly this, although IDLE is smart enough to execute user code in a separate process, so that if it crashes IDLE isn't affected.

3.3.1. Expression Evaluation with eval()

The built-in `eval()` function evaluates a single string as an expression (with access to any global or local context we give it) and returns the result. This is sufficient to build the simple `calculator.py` application that we will review in this subsection. Let's begin by looking at some interaction.

```
$ ./calculator.py
Enter an expression (Ctrl+D to quit): 65
A=65
```

* Parsing, including using PLY and PyParsing, is introduced in this author's book, *Programming in Python 3, Second Edition*; see the Selected Bibliography for details (➤ 287).

```
ANS=65
Enter an expression (Ctrl+D to quit): 72
A=65, B=72
ANS=72
Enter an expression (Ctrl+D to quit): hypotenuse(A, B)
name 'hypotenuse' is not defined
Enter an expression (Ctrl+D to quit): hypot(A, B)
A=65, B=72, C=97.0
ANS=97.0
Enter an expression (Ctrl+D to quit): ^D
```

The user entered two sides of a right-angled triangle and then used the `math.hypot()` function (after making a mistake) to calculate the hypotenuse. After each expression is entered, the `calculator.py` program prints the variables it has created so far (and that are accessible to the user) and the answer to the current expression. (We have indicated user-entered text using bold—with Enter or Return at the end of each line implied—and Ctrl+D with `^D`.)

To make the calculator as convenient as possible, the result of each expression entered is stored in a variable, starting with `A`, then `B`, and so on, and restarting at `A` if `Z` is reached. Furthermore, we have imported all the `math` module's functions and constants (e.g., `hypot()`, `e`, `pi`, `sin()`, etc.) into the calculator's namespace so that the user can access them without qualifying them (e.g., `cos()` rather than `math.cos()`).

If the user enters a string that cannot be evaluated, the calculator prints an error message and then repeats the prompt, and all the existing context is kept intact.

```
def main():
    quit = "Ctrl+Z,Enter" if sys.platform.startswith("win") else "Ctrl+D"
    prompt = "Enter an expression ({} to quit): ".format(quit)
    current = types.SimpleNamespace(letter="A")
    globalContext = global_context()
    localContext = collections.OrderedDict()
    while True:
        try:
            expression = input(prompt)
            if expression:
                calculate(expression, globalContext, localContext, current)
        except EOFError:
            print()
            break
```

We have used EOF (End Of File) to signify that the user has finished. This means that the calculator can be used in a shell pipeline, accepting input redirected from a file, as well as for interactive user input.

We need to keep track of the name of the current variable (A or B or …) so that we can update it each time a calculation is done. However, we can't simply pass it as a string, since strings are copied and cannot be changed. A poor solution is to use a global variable. A better and much more common solution is to create a one-item list; for example, `current = ["A"]`. This list can be passed as `current` and its string can be read or changed by accessing it as `current[0]`.

For this example, we have taken a more modern approach and created a tiny namespace with a single attribute (`letter`) whose value is `"A"`. We can freely pass the `current` simple namespace instance around, and since it has a `letter` attribute, we can read or change the attribute's value using the nice `current.letter` syntax.

The `types.SimpleNamespace` class was introduced in Python 3.3. For earlier versions an equivalent effect can be achieved by writing `current = type("_", (), dict(letter="A"))()`. This creates a new class called `_` with a single attribute called `letter` with an initial value of `"A"`. The built-in `type()` function returns the type of an object if called with one argument, or creates a new class if given a class name, a tuple of base classes, and a dictionary of attributes. If we pass an empty tuple, the base class will be `object`. Since we don't need the class but only an instance, having called `type()`, we immediately call the class itself—hence the extra parentheses—to return the instance of it that we assign to `current`.

Python can supply the current global context using the built-in `globals()` function; this returns a `dict` that we can modify (e.g., add to, as we saw earlier; 23 ◀). Python can also supply the local context using the built-in `locals()` function, although the `dict` returned by this function must *not* be modified.

We want to provide a global context supplemented with the `math` module's constants and functions and an initially empty local context. Although the global context *must* be a `dict`, the local context can be supplied as a `dict`—or as any other mapping object. Here, we have chosen to use a `collections.OrderedDict`—an insertion-ordered dictionary—as the locals context.

Since the calculator can be used interactively, we have created an event loop that is terminated when EOF is encountered. Inside the loop we prompt the user for input (also telling them how to quit), and if they enter any text, we call our `calculate()` function to perform the calculation and to print the results.

```
import math

def global_context():
    globalContext = globals().copy()
    for name in dir(math):
```

```
        if not name.startswith("_"):
            globalContext[name] = getattr(math, name)
    return globalContext
```

This helper function starts by creating a local (shallow-copied) `dict` of the program's global modules, functions, and variables. Then it iterates over all the public constants and functions in the `math` module and, for each one, adds its unqualified name to the `globalContext` `dict` and sets its value to be the actual `math` module constant or function it refers to. So, for example, when the `name` is `"factorial"`, this name is added as a key in the `globalContext`, and its value is set to be the (i.e., a reference to the) `math.factorial()` function. This is what allows the calculator's users to use unqualified names.

A simpler approach would have been to do `from math import *` and then use `globals()` directly, with no need for the `globalContext` `dict`. Such an approach is probably okay for the `math` module, but the way we have done it here provides finer control that might be more appropriate for other modules.

```
def calculate(expression, globalContext, localContext, current):
    try:
        result = eval(expression, globalContext, localContext)
        update(localContext, result, current)
        print(", ".join(["{}={}".format(variable, value)
                for variable, value in localContext.items()]))
        print("ANS={}".format(result))
    except Exception as err:
        print(err)
```

This is the function where we ask Python to evaluate the string expression using the global and local context dictionaries that we have created. If the `eval()` succeeds, we update the local context with the result and print the variables and the result. And if an exception occurs, we safely print it. Since we used a `collections.OrderedDict` for the local context, the `items()` method returns the items in insertion order without the need for an explicit sort. (Had we used a plain `dict` we would have needed to write `sorted(localContext.items())`.)

Although it is usually poor practice to use the `Exception` catch-all exception, it seems reasonable in this case, because the user's expression could raise any kind of exception at all.

```
def update(localContext, result, current):
    localContext[current.letter] = result
    current.letter = chr(ord(current.letter) + 1)
    if current.letter > "Z": # We only support 26 variables
        current.letter = "A"
```

This function assigns the result to the next variable in the cyclic sequence A … Z A … Z …. This means that after the user has entered 26 expressions, the result of the last one is set as Z's value, and the result of the next one will overwrite A's value, and so on.

The `eval()` function will evaluate *any* Python expression. This is potentially dangerous if the expression is received from an untrusted source. An alternative is to use the standard library's more restrictive—and safe—`ast.literal_eval()` function.

3.3.2. Code Evaluation with exec()

The built-in `exec()` function can be used to execute arbitrary pieces of Python code. Unlike `eval()`, `exec()` is not restricted to a single expression and always returns `None`. Context can be passed to `exec()` in the same way as for `eval()`, via globals and locals dictionaries. Results can be retrieved from `exec()` through the local context it is passed.

In this subsection, we will review the `genome1.py` program. This program creates a `genome` variable (a string of random A, C, G, and T letters) and executes eight pieces of user code with the genome in the code's context.

```
context = dict(genome=genome, target="G[AC]{2}TT", replace="TCGA")
execute(code, context)
```

This code snippet shows the creation of the `context` dictionary with some data for the user's code to work on and the execution of a user's `Code` object (`code`) with the given context.

```
TRANSFORM, SUMMARIZE = ("TRANSFORM", "SUMMARIZE")

Code = collections.namedtuple("Code", "name code kind")
```

We expect user code to be provided in the form of `Code` named tuples, with a descriptive name, the code itself (as a string), and a kind—either `TRANSFORM` or `SUMMARIZE`. When executed, the user code should create either a `result` object or an `error` object. If their code's kind is `TRANSFORM`, the `result` is expected to be a new genome string, and if the kind is `SUMMARIZE`, `result` is expected to be a number. Naturally, we will try to make our code robust enough to cope with user code that doesn't meet these requirements.

```
def execute(code, context):
    try:
        exec(code.code, globals(), context)
        result = context.get("result")
        error = context.get("error")
```