B_controlflow

March 1, 2019

This is a Jupyter Notebook adaptation of the Python 3 tutorial[2].

1 More Control Flow Tools

Besides the while statement just introduced, Python knows the usual control flow statements known from other languages, with some twists.

1.1 if Statements

Perhaps the most well-known statement type is the if statement. For example:

```
In [ ]: x = int(input("Please enter an integer: "))

In [ ]: if x < 0:
   x = 0
   print('Negative changed to zero')
elif x == 0:
   print('Zero')
elif x == 1:
   print('Single')
else:
   print('More')
```

There can be zero or more elif parts, and the else part is optional. The keyword elif is short for else if, and is useful to avoid excessive indentation. An if ... elif ... elif ... sequence is a substitute for the switch or case statements found in other languages.

1.2 for Statements

The for statement in Python differs a bit from what you may be used to in C or Pascal. Rather than always iterating over an arithmetic progression of numbers (like in Pascal), or giving the user the ability to define both the iteration step and halting condition (as C), Python’s for statement iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence. For example (no pun intended):

```
In [ ]: # Measure some strings:
   words = ['cat', 'window', 'defenestrate']
```
In [ ]: for w in words:
    print(w, len(w))

If you need to modify the sequence you are iterating over while inside the loop (for example to
duplicate selected items), it is recommended that you first make a copy. Iterating over a sequence
does not implicitly make a copy. The slice notation makes this especially convenient:

In [ ]: for w in words[:]:  # Loop over a slice copy of the entire list.
    if len(w) > 6:
        words.insert(0, w)

In [ ]: words

1.3 The range Function

If you do need to iterate over a sequence of numbers, the built-in function range comes in handy.
It generates arithmetic progressions:

In [ ]: for i in range(5):
    print(i)

The given end point is never part of the generated sequence; range(10) generates 10 values,
the legal indices for items of a sequence of length 10. It is possible to let the range start at another
number, or to specify a different increment (even negative; sometimes this is called the 'step'):

range(5, 10)
  5 through 9

range(0, 10, 3)
  0, 3, 6, 9

range(-10, -100, -30)
  -10, -40, -70

To iterate over the indices of a sequence, you can combine range and len as follows:

In [ ]: a = ['Mary', 'had', 'a', 'little', 'lamb']

In [ ]: for i in range(len(a)):
    print(i, a[i])

In most such cases, however, it is convenient to use the enumerate function.
A strange thing happens if you just print a range:

In [ ]: print(range(10))

In many ways the object returned by range behaves as if it is a list, but in fact it isn’t. It is an
object which returns the successive items of the desired sequence when you iterate over it, but it
doesn’t really make the list, thus saving space.

We say such an object is iterable, that is, suitable as a target for functions and constructs that
expect something from which they can obtain successive items until the supply is exhausted. We
have seen that the for statement is such an iterator. The function list is another; it creates lists
from iterables:
In [ ]: \text{list(range}(5))

Later we will see more functions that return iterables and take iterables as argument.

1.4 \text{break and continue Statements, and else Clauses on Loops}

The \text{break} statement, like in C, breaks out of the innermost enclosing \text{for} or \text{while} loop.

Loop statements may have an \text{else} clause; it is executed when the loop terminates through exhaustion of the list (with \text{for}) or when the condition becomes false (with \text{while}), but not when the loop is terminated by a \text{break} statement. This is exemplified by the following loop, which searches for prime numbers:

In [ ]: \text{for n in range}(2, 10):
   \text{for x in range}(2, n):
      \text{if n} \% x == 0:
         \text{print(n, 'equals', x, '*', n//x)}
         \text{break}
      \text{else}:
         \# loop fell through without finding a factor
         \text{print(n, 'is a prime number')}

(Yes, this is the correct code. Look closely: the \text{else} clause belongs to the \text{for} loop, \text{not} the \text{if} statement.)

When used with a loop, the \text{else} clause has more in common with the \text{else} clause of a \text{try} statement than it does that of \text{if} statements: a \text{try} statement’s \text{else} clause runs when no exception occurs, and a loop’s \text{else} clause runs when no \text{break} occurs.

The \text{continue} statement, also borrowed from C, continues with the next iteration of the loop:

In [ ]: \text{for num in range}(2, 10):
   \text{if num} \% 2 == 0:
      \text{print("Found an even number", num)}
      \text{continue}
      \text{print("Found a number", num)}

1.5 \text{pass Statements}

The \text{pass} statement does nothing. It can be used when a statement is required syntactically but the program requires no action. For example: (this example is not a notebook cell because it would live-lock the ipython kernel, it can be tried by starting the interpreter on the command line, where it can be stopped with Ctrl+C)

while True:
   \text{pass}  \# Busy-wait for keyboard interrupt (Ctrl+C)

This is commonly used for creating minimal classes:

In [ ]: \text{class MyEmptyClass:}
   \text{pass}
Another place pass can be used is as a place-holder for a function or conditional body when you are working on new code, allowing you to keep thinking at a more abstract level. The pass is silently ignored:

```
In [ ]: def initlog(*args):
    pass  # Remember to implement this!
```

### 1.6 Defining Functions

We can create a function that writes the Fibonacci series to an arbitrary boundary:

```
In [ ]: def fib(n):
    # write Fibonacci series up to n
    """Print a Fibonacci series up to n.""
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()
```

```
In [ ]: # Now call the function we just defined:
    fib(2000)
```

The keyword def introduces a function definition. It must be followed by the function name and the parenthesized list of formal parameters. The statements that form the body of the function start at the next line, and must be indented.

The first statement of the function body can optionally be a string literal; this string literal is the function’s documentation string, or docstring. There are tools which use docstrings to automatically produce online or printed documentation, or to let the user interactively browse through code; it’s good practice to include docstrings in code that you write, so make a habit of it.

The execution of a function introduces a new symbol table used for the local variables of the function. More precisely, all variable assignments in a function store the value in the local symbol table; whereas variable references first look in the local symbol table, then in the local symbol tables of enclosing functions, then in the global symbol table, and finally in the table of built-in names. Thus, global variables cannot be directly assigned a value within a function (unless named in a global statement), although they may be referenced.

The actual parameters (arguments) to a function call are introduced in the local symbol table of the called function when it is called; thus, arguments are passed using call by value (where the value is always an object reference, not the value of the object).[^1] When a function calls another function, a new local symbol table is created for that call.

A function definition introduces the function name in the current symbol table. The value of the function name has a type that is recognized by the interpreter as a user-defined function. This value can be assigned to another name which can then also be used as a function. This serves as a general renaming mechanism:

```
In [ ]: fib

In [ ]: f = fib
    f(100)
```

[^1]: Call by reference (call by value) is a parameter passing method in which the function receives a reference to the variable, not the value of the variable.
Coming from other languages, you might object that fib is not a function but a procedure since it doesn’t return a value. In fact, even functions without a return statement do return a value, albeit a rather boring one. This value is called None (it’s a built-in name). Writing the value None is normally suppressed by the interpreter if it would be the only value written. You can see it if you really want to using print:

In [ ]: fib(0)

In [ ]: print(fib(0))

It is simple to write a function that returns a list of the numbers of the Fibonacci series, instead of printing it:

In [ ]: def fib2(n):
    """Return a list containing the Fibonacci series up to n."""
    result = []
    a, b = 0, 1
    while a < n:
        result.append(a)  # see below
        a, b = b, a+b
    return result

In [ ]: f100 = fib2(100)  # call it
f100  # write the result

This example, as usual, demonstrates some new Python features:

• The return statement returns with a value from a function. return without an expression argument returns None. Falling off the end of a function also returns None.

• The statement result.append(a) calls a method of the list object result. A method is a function that ‘belongs’ to an object and is named obj.methodname, where obj is some object (this may be an expression), and methodname is the name of a method that is defined by the object’s type. Different types define different methods. Methods of different types may have the same name without causing ambiguity. (It is possible to define your own object types and methods, using classes) The method append shown in the example is defined for list objects; it adds a new element at the end of the list. In this example it is equivalent to result = result + [a], but more efficient.

1.7 More on Defining Functions

It is also possible to define functions with a variable number of arguments. There are three forms, which can be combined.

1.7.1 Default Argument Values

The most useful form is to specify a default value for one or more arguments. This creates a function that can be called with fewer arguments than it is defined to allow. For example:
In [
]:
def ask_ok(prompt, retries=4, reminder='Please try again!'):
    while True:
        ok = input(prompt)
        if ok in ('y', 'ye', 'yes'):
            return True
        if ok in ('n', 'no', 'nop', 'nope'):
            return False
        retries = retries - 1
        if retries < 0:
            raise ValueError('invalid user response')
        print(reminder)

This function can be called in several ways:

• giving only the mandatory argument:

In [ ]: ask_ok('Do you really want to quit?')

• giving one of the optional arguments:

In [ ]: ask_ok('OK to overwrite the file?', 2)

• or even giving all arguments:

In [ ]: ask_ok('OK to overwrite the file?', 2, 'Come on, only yes or no!')

This example also introduces the in keyword. This tests whether or not a sequence contains a certain value.

The default values are evaluated at the point of function definition in the defining scope, so that:

i = 5

def f(arg=i):
    print(arg)

i = 6
f()

will print 5.

**Important warning:** The default value is evaluated only once. This makes a difference when
the default is a mutable object such as a list, dictionary, or instances of most classes. For example,
the following function accumulates the arguments passed to it on subsequent calls:

def f(a, L=[]):
    L.append(a)
    return L

print(f(1))
pfint(f(2))
pfint(f(3))
This will print:

```
[1]
[1, 2]
[1, 2, 3]
```

If you don’t want the default to be shared between subsequent calls, you can write the function like this instead:

```python
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L
```

### 1.7.2 Keyword Arguments

Functions can also be called using keyword arguments of the form `kwarg=value`. For instance, the following function:

```python
def parrot(voltage, state='a stiff', action='voom', type='Norwegian Blue'):
    print("-- This parrot wouldn't", action, end=' ')  
    print("if you put", voltage, "volts through it.")
    print("-- Lovely plumage, the", type)
    print("-- It's", state, "!")
```

accepts one required argument (`voltage`) and three optional arguments (`state`, `action`, and `type`). This function can be called in any of the following ways:

- `parrot(1000) # 1 positional argument`
- `parrot(voltage=1000) # 1 keyword argument`
- `parrot(voltage=1000000, action='VOOOOOM') # 2 keyword arguments`
- `parrot(action='VOOOOOM', voltage=1000000) # 2 keyword arguments`
- `parrot('a million', 'bereft of life', 'jump') # 3 positional arguments`
- `parrot('a thousand', state='pushing up the daisies') # 1 positional, 1 keyword`

but all the following calls would be invalid:

- `parrot() # required argument missing`
- `parrot(voltage=5.0, 'dead') # non-keyword argument after a keyword argument`
- `parrot(110, voltage=220) # duplicate value for the same argument`
- `parrot(actor='John Cleese') # unknown keyword argument`

In a function call, keyword arguments must follow positional arguments. All the keyword arguments passed must match one of the arguments accepted by the function (e.g. `actor` is not a valid argument for the `parrot` function), and their order is not important. This also includes non-optional arguments (e.g. `parrot(voltage=1000) is valid too). No argument may receive a value more than once. Here’s an example that fails due to this restriction:
```
In [ ]: def function(a):
    pass

In [ ]: function(0, a=0)
```

When a final formal parameter of the form `**name` is present, it receives a dictionary (see `typesmapping`) containing all keyword arguments except for those corresponding to a formal parameter. This may be combined with a formal parameter of the form `*name` (described in the next subsection) which receives a tuple containing the positional arguments beyond the formal parameter list. (*name must occur before **name.*) For example, if we define a function like this:

```python
def cheeseshop(kind, *arguments, **keywords):
    print("-- Do you have any", kind, "?"
    print("-- I'm sorry, we're all out of", kind)
    for arg in arguments:
        print(arg)
    print("-" * 40)
    keys = sorted(keywords.keys())
    for kw in keys:
        print(kw, ":", keywords[kw])
```

It could be called like this:

```python
cheeseshop("Limburger", "It's very runny, sir.",
       "It's really very, VERY runny, sir.",
       shopkeeper="Michael Palin",
       client="John Cleese",
       sketch="Cheese Shop Sketch")
```

and of course it would print:

```
In [ ]: -- Do you have any Limburger?
    -- I'm sorry, we're all out of Limburger
    It's very runny, sir.
    It's really very, VERY runny, sir.
    -----------------------------
    client : John Cleese
    shopkeeper : Michael Palin
    sketch : Cheese Shop Sketch
```

Note that the list of keyword argument names is created by sorting the result of the keywords dictionary's `keys()` method before printing its contents; if this is not done, the order in which the arguments are printed is undefined.

1.7.3 Arbitrary Argument Lists

Finally, the least frequently used option is to specify that a function can be called with an arbitrary number of arguments. These arguments will be wrapped up in a tuple. Before the variable number of arguments, zero or more normal arguments may occur:
def write_multiple_items(file, separator, *args):
    file.write(separator.join(args))

    Normally, these variadic arguments will be last in the list of formal parameters, because they scoop up all remaining input arguments that are passed to the function. Any formal parameters which occur after the *args parameter are 'keyword-only' arguments, meaning that they can only be used as keywords rather than positional arguments. :

    In [ ]: def concat(*args, sep="/"):  
            return sep.join(args)

    In [ ]: concat("earth", "mars", "venus")

    In [ ]: concat("earth", "mars", "venus", sep=".")

1.7.4 Unpacking Argument Lists

The reverse situation occurs when the arguments are already in a list or tuple but need to be unpacked for a function call requiring separate positional arguments. For instance, the built-in range function expects separate start and stop arguments. If they are not available separately, write the function call with the *-operator to unpack the arguments out of a list or tuple:

    In [ ]: list(range(3, 6))  
            # normal call with separate arguments

    In [ ]: args = [3, 6]  
            list(range(*args))  
            # call with arguments unpacked from a list

    In the same fashion, dictionaries can deliver keyword arguments with the **-operator:

    In [ ]: def parrot(voltage, state='a stiff', action='voom'):  
            print("-- This parrot wouldn't", action, end='')
            print("if you put", voltage, "volts through it.", end='')
            print("E's", state, "!")

    In [ ]: d = {"voltage": "four million", "state": "bleedin' demised", "action": "VOOM"}
    parrot(**d)

1.7.5 Lambda Expressions

Small anonymous functions can be created with the lambda keyword. This function returns the sum of its two arguments: lambda a, b: a+b. Lambda functions can be used wherever function objects are required. They are syntactically restricted to a single expression. Semantically, they are just syntactic sugar for a normal function definition. Like nested function definitions, lambda functions can reference variables from the containing scope:

    In [ ]: def make_incrementor(n):
            return lambda x: x + n

    In [ ]: f = make_incrementor(42)

    f(0)
The above example uses a lambda expression to return a function. Another use is to pass a small function as an argument:

```
In [ ]: pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
    pairs.sort(key=lambda pair: pair[1])

In [ ]: pairs
```

### 1.7.6 Documentation Strings

Here are some conventions about the content and formatting of documentation strings.

The first line should always be a short, concise summary of the object’s purpose. For brevity, it should not explicitly state the object’s name or type, since these are available by other means (except if the name happens to be a verb describing a function’s operation). This line should begin with a capital letter and end with a period.

If there are more lines in the documentation string, the second line should be blank, visually separating the summary from the rest of the description. The following lines should be one or more paragraphs describing the object’s calling conventions, its side effects, etc.

The Python parser does not strip indentation from multi-line string literals in Python, so tools that process documentation have to strip indentation if desired. This is done using the following convention. The first non-blank line after the first line of the string determines the amount of indentation for the entire documentation string. (We can’t use the first line since it is generally adjacent to the string’s opening quotes so its indentation is not apparent in the string literal.) Whitespace “equivalent” to this indentation is then stripped from the start of all lines of the string. Lines that are indented less should not occur, but if they occur all their leading whitespace should be stripped. Equivalence of whitespace should be tested after expansion of tabs (to 8 spaces, normally).

Here is an example of a multi-line docstring:

```
In [ ]: def my_function():
    """Do nothing, but document it.
    No, really, it doesn't do anything.
    ""
    pass

In [ ]: print(my_function.__doc__)
```

### 1.7.7 Function Annotations

Function annotations are completely optional metadata information about the types used by user-defined functions (see 484 for more information).

Annotations are stored in the `__annotations__` attribute of the function as a dictionary and have no effect on any other part of the function. Parameter annotations are defined by a colon after the parameter name, followed by an expression evaluating to the value of the annotation. Return annotations are defined by a literal ->, followed by an expression, between the parameter list and the colon denoting the end of the `def` statement. The following example has a positional argument, a keyword argument, and the return value annotated:
```python
In [ ]: def f(ham: str, eggs: str = 'eggs') -> str:
    print("Annotations:", f.__annotations__)
    print("Arguments:", ham, eggs)
    return ham + ' and ' + eggs

In [ ]: f('spam')
```

## 1.8 Intermezzo: Coding Style

Now that you are about to write longer, more complex pieces of Python, it is a good time to talk about coding style. Most languages can be written (or more concise, formatted) in different styles; some are more readable than others. Making it easy for others to read your code is always a good idea, and adopting a nice coding style helps tremendously for that.

For Python, [8] has emerged as the style guide that most projects adhere to; it promotes a very readable and eye-pleasing coding style. Every Python developer should read it at some point; here are the most important points extracted for you:

- Use 4-space indentation, and no tabs.
  
  4 spaces are a good compromise between small indentation (allows greater nesting depth) and large indentation (easier to read). Tabs introduce confusion, and are best left out.

- Wrap lines so that they don’t exceed 79 characters.

This helps users with small displays and makes it possible to have several code files side-by-side on larger displays.

- Use blank lines to separate functions and classes, and larger blocks of code inside functions.
- When possible, put comments on a line of their own.
- Use docstrings.
- Use spaces around operators and after commas, but not directly inside bracketing constructs:
  ```python
  a = f(1, 2) + g(3, 4).
  ```
- Name your classes and functions consistently; the convention is to use CamelCase for classes and lower_case_with_underscores for functions and methods. Always use self as the name for the first method argument.
- Don’t use fancy encodings if your code is meant to be used in international environments. Python’s default, UTF-8, or even plain ASCII work best in any case.
- Likewise, don’t use non-ASCII characters in identifiers if there is only the slightest chance people speaking a different language will read or maintain the code.

Footnotes