Discrete Mathematics Course Note — Getting started with *Python*

Tyng–Ruey Chuang

2011-05-03

- We will use Python 2.7.1 which is available at `http://www.python.org/` . Here is a brief guide.
  - Download and install Python 2.7.1:
    `http://www.python.org/getit/`
  - Consult Python 2.7.1 documentation on–line:
    `http://docs.python.org/download.html`

    Or, download an off–line folder of Python 2.7.1 documentation:
    `http://docs.python.org/download.html`
  - Start with *Python Tutorial: Release 2.7.1*:
    `http://docs.python.org/tutorial/index.html`

    Or, browse the pdf file `tutorial.pdf` in the off–line folder of documentation.
- Most often you run Python as an interpreter from the command–line. Here is a brief demo.
  - Type `python` after the command–line prompt (don't forget the `return` key), you will get:

    ```
    Python 2.7.1 (r271:86882M, Nov 30 2010, 09:39:13)
    [GCC 4.0.1 (Apple Inc. build 5494)] on darwin
    Type "help", "copyright", "credits" or "license" for more information.
    >>>
    ```

    Type `print 1+1` after the prompt `>>>` (again, don't forget the `return` key), you get:

    ```
    >>> print 1+1
    2
    >>>
    ```
  - You get out of python by typing *control–D*. That will get you back to the command–line.
- Study the code in the file `ex1.py` (yes, files for Python program end with suffix `.py`).
- You can execute Python code in a file, say, `ex1.py`, directly from the command–line:

  ```
  python ex1.py
  ```
- You can also import Python code in a file, say, `ex1.py`, to the Python interpreter:

  ```
  >>> import ex1
  ```
- Study the code in the file `ex2.py`, and import the code to the Python interpreter:

  ```
  >>> import ex2
  ```
- File `ex2.py` defines several functions. One of them is `fib(n)` for computing the n^{th} Fibonacci number. This function can be invoked by:

  ```
  >>> ex2.fib(10)
  55
  ```
- Note that you can use the *up–arrow* key to retrieve expressions you had typed in the Python interpreter. Try for yourself the *left–arrow*, *right–arrow*, and *delete* keys too!
- Python uses indentation to determine the grouping of statements. You need to use spaces and tabs at the beginning of a line for the correct indentation of Python statements.
- Read the tutorial, you are on your own now!
- Of course, you can always ask me and the TA questions by e-mail!

```
 1 print 0
 2
 3 print 1 + 2 + 3
 4 print (1 + 2) + 3
 5 print 1 + (2 + 3)
 6
 7 print 3 - 2 - 1
 8 print (3 - 2) - 1
 9 print 3 - (2 - 1)
10
11 print 1 + 2 * 3
12 print 1 + (2 * 3)
13 print (1 + 2) * 3
14
15 print 8 / 4 / 2
16 print (8 / 4) / 2
17 print 8 / (4 / 2)
18
19 print 365 * 24 * 60 * 60
20
21 print 0, 1, 2, 3
22
23 print (1 + 2 + 3), (3 - 2 - 1), (1 + 2 * 3), (8 / 4 / 2)
24
25 print 1 + 2 + 3, 3 - 2 - 1, 1 + 2 * 3, 8 / 4 / 2
26
27 print "This is a sentence."
28
29 print "This", "is", "anothere", "sentence."
30
31 print "This " "is " "yet " "another " "sentence."
32
33 print "One year has 365 days."
34
35 print "One year has", 365, "days."
36
37 print 365, "days have",  (365 * 24 * 60 * 60), "seconds."
38
39 print 365, "days have",  (365 * 24 * 60 * 60), "seconds."
40
```

```
 1 print range (1, 20, 2)
 2 print range (1, 20)
 3 print range (20)
 4
 5 a, b, n = 0, 1, 1
 6 while b < 10000:
 7     print "fib(",  n, ") = ", b
 8     a, b = b, a+b
 9     n = n+1
10
11 print "----- end of part 0 ----"
12
13 def fib (n):
14     if n <= 2:
15         return 1
16     else:
17         return fib(n-1) + fib(n-2)
18
19 for n in range (1, 20):
20     print "fib(",  n, ") = ", fib(n)
21
22 print "----- end of part 1 ----"
23
24 def fib1 (n):
25     a, b, k = 0, 1, 1
26     while k < n:
27         a, b = b, a+b
28         k = k + 1
29     return b
30
31 for n in range (1, 20):
32     print "fib(", n, ") = ", fib1(n)
33
34 print "----- end of part 2 ----"
35
36 def fib3 (n, a, b):
37     if n <= 1:
38         return a
39     elif n == 2:
40         return b
41     else:
42         return fib3(n-1, b, a+b)
43
44 def fib2(n):
45     return fib3(n, 1, 1)
46
47 for n in range (1, 20):
48     print "fib(", n, ") = ", fib2(n)
49
50 print "----- end of part 3 ----"
```

Python Tutorial

Release 2.7.1

Guido van Rossum
Fred L. Drake, Jr., editor

May 01, 2011

Python Software Foundation
Email: docs@python.org

CHAPTER

THREE

AN INFORMAL INTRODUCTION TO PYTHON

In the following examples, input and output are distinguished by the presence or absence of prompts (>>> and ...): to repeat the example, you must type everything after the prompt, when the prompt appears; lines that do not begin with a prompt are output from the interpreter. Note that a secondary prompt on a line by itself in an example means you must type a blank line; this is used to end a multi-line command.

Many of the examples in this manual, even those entered at the interactive prompt, include comments. Comments in Python start with the hash character, #, and extend to the end of the physical line. A comment may appear at the start of a line or following whitespace or code, but not within a string literal. A hash character within a string literal is just a hash character. Since comments are to clarify code and are not interpreted by Python, they may be omitted when typing in examples.

Some examples:

```
# this is the first comment
SPAM = 1                 # and this is the second comment
                         # ... and now a third!
STRING = "# This is not a comment."
```

3.1 Using Python as a Calculator

Let's try some simple Python commands. Start the interpreter and wait for the primary prompt, >>>. (It shouldn't take long.)

3.1.1 Numbers

The interpreter acts as a simple calculator: you can type an expression at it and it will write the value. Expression syntax is straightforward: the operators +, −, * and / work just like in most other languages (for example, Pascal or C); parentheses can be used for grouping. For example:

```
>>> 2+2
4
>>> # This is a comment
... 2+2
4
>>> 2+2  # and a comment on the same line as code
4
>>> (50-5*6)/4
5
>>> # Integer division returns the floor:
... 7/3
2
```

```
>>> 7/-3
-3
```

The equal sign (`'='`) is used to assign a value to a variable. Afterwards, no result is displayed before the next interactive prompt:

```
>>> width = 20
>>> height = 5*9
>>> width * height
900
```

A value can be assigned to several variables simultaneously:

```
>>> x = y = z = 0  # Zero x, y and z
>>> x
0
>>> y
0
>>> z
0
```

Variables must be "defined" (assigned a value) before they can be used, or an error will occur:

```
>>> # try to access an undefined variable
... n
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

There is full support for floating point; operators with mixed type operands convert the integer operand to floating point:

```
>>> 3 * 3.75 / 1.5
7.5
>>> 7.0 / 2
3.5
```

Complex numbers are also supported; imaginary numbers are written with a suffix of `j` or `J`. Complex numbers with a nonzero real component are written as `(real+imagj)`, or can be created with the `complex(real, imag)` function.

```
>>> 1j * 1J
(-1+0j)
>>> 1j * complex(0,1)
(-1+0j)
>>> 3+1j*3
(3+3j)
>>> (3+1j)*3
(9+3j)
>>> (1+2j)/(1+1j)
(1.5+0.5j)
```

Complex numbers are always represented as two floating point numbers, the real and imaginary part. To extract these parts from a complex number *z*, use `z.real` and `z.imag`.

```
>>> a=1.5+0.5j
>>> a.real
1.5
>>> a.imag
0.5
```

The conversion functions to floating point and integer (`float()`, `int()` and `long()`) don't work for complex numbers — there is no one correct way to convert a complex number to a real number. Use `abs(z)` to get its magnitude (as a float) or `z.real` to get its real part.

```
>>> a = ['spam', 'eggs', 100, 1234]
>>> a
['spam', 'eggs', 100, 1234]
```

Like string indices, list indices start at 0, and lists can be sliced, concatenated and so on:

```
>>> a[0]
'spam'
>>> a[3]
1234
>>> a[-2]
100
>>> a[1:-1]
['eggs', 100]
>>> a[:2] + ['bacon', 2*2]
['spam', 'eggs', 'bacon', 4]
>>> 3*a[:3] + ['Boo!']
['spam', 'eggs', 100, 'spam', 'eggs', 100, 'spam', 'eggs', 100, 'Boo!']
```

All slice operations return a new list containing the requested elements. This means that the following slice returns a shallow copy of the list *a*:

```
>>> a[:]
['spam', 'eggs', 100, 1234]
```

Unlike strings, which are *immutable*, it is possible to change individual elements of a list:

```
>>> a
['spam', 'eggs', 100, 1234]
>>> a[2] = a[2] + 23
>>> a
['spam', 'eggs', 123, 1234]
```

Assignment to slices is also possible, and this can even change the size of the list or clear it entirely:

```
>>> # Replace some items:
... a[0:2] = [1, 12]
>>> a
[1, 12, 123, 1234]
>>> # Remove some:
... a[0:2] = []
>>> a
[123, 1234]
>>> # Insert some:
... a[1:1] = ['bletch', 'xyzzy']
>>> a
[123, 'bletch', 'xyzzy', 1234]
>>> # Insert (a copy of) itself at the beginning
>>> a[:0] = a
>>> a
[123, 'bletch', 'xyzzy', 1234, 123, 'bletch', 'xyzzy', 1234]
>>> # Clear the list: replace all items with an empty list
>>> a[:] = []
>>> a
[]
```

The built-in function `len()` also applies to lists:

```
>>> a = ['a', 'b', 'c', 'd']
>>> len(a)
4
```

It is possible to nest lists (create lists containing other lists), for example:

```
>>> q = [2, 3]
>>> p = [1, q, 4]
>>> len(p)
3
>>> p[1]
[2, 3]
>>> p[1][0]
2
>>> p[1].append('xtra')      # See section 5.1
>>> p
[1, [2, 3, 'xtra'], 4]
>>> q
[2, 3, 'xtra']
```

Note that in the last example, `p[1]` and `q` really refer to the same object! We'll come back to *object semantics* later.

3.2 First Steps Towards Programming

Of course, we can use Python for more complicated tasks than adding two and two together. For instance, we can write an initial sub-sequence of the *Fibonacci* series as follows:

```
>>> # Fibonacci series:
... # the sum of two elements defines the next
... a, b = 0, 1
>>> while b < 10:
...     print b
...     a, b = b, a+b
...
1
1
2
3
5
8
```

This example introduces several new features.

- The first line contains a *multiple assignment*: the variables `a` and `b` simultaneously get the new values 0 and 1. On the last line this is used again, demonstrating that the expressions on the right-hand side are all evaluated first before any of the assignments take place. The right-hand side expressions are evaluated from the left to the right.
- The `while` loop executes as long as the condition (here: `b < 10`) remains true. In Python, like in C, any non-zero integer value is true; zero is false. The condition may also be a string or list value, in fact any sequence; anything with a non-zero length is true, empty sequences are false. The test used in the example is a simple comparison. The standard comparison operators are written the same as in C: < (less than), > (greater than), == (equal to), <= (less than or equal to), >= (greater than or equal to) and `!=` (not equal to).
- The *body* of the loop is *indented*: indentation is Python's way of grouping statements. Python does not (yet!) provide an intelligent input line editing facility, so you have to type a tab or space(s) for each indented line. In practice you will prepare more complicated input for Python with a text editor; most text editors have an auto-indent facility. When a compound statement is entered interactively, it must be followed by a blank line to indicate completion (since the parser cannot guess when you have typed the last line). Note that each line within a basic block must be indented by the same amount.
- The `print` statement writes the value of the expression(s) it is given. It differs from just writing the expression you want to write (as we did earlier in the calculator examples) in the way it handles multiple expressions and strings. Strings are printed without quotes, and a space is inserted between items, so you can format things nicely, like this:

```
>>> i = 256*256
>>> print 'The value of i is', i
The value of i is 65536
```

A trailing comma avoids the newline after the output:

```
>>> a, b = 0, 1
>>> while b < 1000:
...     print b,
...     a, b = b, a+b
...
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

Note that the interpreter inserts a newline before it prints the next prompt if the last line was not completed.

CHAPTER

FOUR

MORE CONTROL FLOW TOOLS

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

4.1 `if` Statements

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

```
>>> x = int(raw_input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...      x = 0
...      print 'Negative changed to zero'
... elif x == 0:
...      print 'Zero'
... elif x == 1:
...      print 'Single'
... else:
...      print 'More'
...
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.

4.2 `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):

```
>>> # Measure some strings:
... a = ['cat', 'window', 'defenestrate']
>>> for x in a:
...     print x, len(x)
...
cat 3
window 6
defenestrate 12
```

It is not safe to modify the sequence being iterated over in the loop (this can only happen for mutable sequence types, such as lists). If you need to modify the list you are iterating over (for example, to duplicate selected items) you must iterate over a copy. The slice notation makes this particularly convenient:

```
>>> for x in a[:]: # make a slice copy of the entire list
...     if len(x) > 6: a.insert(0, x)
...
>>> a
['defenestrate', 'cat', 'window', 'defenestrate']
```

4.3 The `range()` Function

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

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

The given end point is never part of the generated list; `range(10)` generates a list of 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, 6, 7, 8, 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:

```
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print i, a[i]
...
0 Mary
1 had
2 a
3 little
4 lamb
```

In most such cases, however, it is convenient to use the `enumerate()` function, see *Looping Techniques*.

4.4 `break` and `continue` Statements, and `else` Clauses on Loops

The `break` statement, like in C, breaks out of the smallest enclosing `for` or `while` loop.

The `continue` statement, also borrowed from C, continues with the next iteration of the loop.

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

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print n, 'equals', x, '*', n/x
...             break
...     else:
...         # loop fell through without finding a factor
```

```
...         print n, 'is a prime number'
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

4.5 pass Statements

The pass statement does nothing. It can be used when a statement is required syntactically but the program requires no action. For example:

```
>>> while True:
...     pass  # Busy-wait for keyboard interrupt (Ctrl+C)
...
```

This is commonly used for creating minimal classes:

```
>>> class MyEmptyClass:
...     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:

```
>>> def initlog(*args):
...     pass   # Remember to implement this!
...
```

4.6 Defining Functions

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

```
>>> 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,
...         a, b = b, a+b
...
>>> # Now call the function we just defined:
... fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

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*. (More about docstrings can be found in the section *Documentation Strings*.) 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.