CS313E: Elements of Software Design

Basic Python

Dr. Bill Young
Department of Computer Sciences
University of Texas at Austin

Last updated: September 11, 2012 at 13:16

Acknowledgement

Much of the material in this introductory slideset is from Dr. Shyamal Mitra, and used with his blessing.

Introduction to Python

Python is a simple but powerful scripting language. It was developed by Guido van Rossum in the Netherlands in the late 1980s. The language takes its name from the British comedy series *Monty Python's Flying Circus*. Python is an interpreted language unlike C or Fortran that are compiled languages.

- Clean syntax that is concise. You can say a lot more with fewer words.
- Design is compact. You can carry all the language constructs in your head.
- There is a very powerful library of useful functions available.

This means that you can be productive quite easily and fast. You will be spending more time solving problems and writing code than grappling with the idiosyncrasies of the language.

Running Python: Interactively

There are several ways to run Python. One is to start Python and run programs in the *interactive loop*.

```
> python3
>>> print ("Hello World!")
Hello World!
```

This uses the *read-execute-print* loop.

1. Read in a legal Python form.
2. Execute the form.
3. Print the result to standard output.
4. Go to step 1.

Running Python: From File

For longer programs you can store your program in a file that ends with a `.py` extension. Example: saved in a file called `Hello.py` is the single statement: `print ("Hello World!")`
To run your program, type at the command line

```
> python Hello.py
```

You can also create a stand alone script. On a Unix / Linux machine you can create a *script* called `Hello.py` containing the following lines (assuming that's where your Python implementation lives):

```
#!/lusr/bin/python
print ("Hello World!")
```

Running Python: From File

Let's try running this:

```
> Hello.py
Hello.py: Permission denied.
```

What went wrong?
Before you run it, you must tell Linux that it's an *executable* file:

```
> chmod +x Hello.py
```

Then run it:

```
> Hello.py
Hello World!
```

Structure of a Computer Language

- Character set
- Variables
- Types
- Operators
- Assignments
- Conditionals
- Loops
- Functions
- Arrays
- Input / Output

Basic Syntax: Character Set

Python uses the traditional ASCII character set. Recent versions also recognize the Unicode character set. The ASCII character set is a subset of the Unicode character set.

Python *does not have a separate character type.* A character is represented as a string containing one item (e.g., `"a"`).

Identifiers

Python is case sensitive. These are rules for creating an indentifier:

- Must start with a letter or underscore (_)
- Can be followed by any number of letters, digits, or underscores.
- Cannot be a reserved word.

Python reserved words are:

False, None, True, and, as, assert, break, class, continue, def, del, elif, else, except, finally, for, from, global, if, import, in, is, lambda, nonlocal, not, or, pass, raise, return, try, while, with, yield.

Variables

A *variable* in Python denotes a memory location. In that location is the address where the *value* is stored in memory. Consider this assignment:

```
x = 1
```

A memory location is set aside for the variable `x`. The value `1` *is stored in another place in memory*. The address of that location is stored in the memory location denoted by `x`.

This simple observation explains a lot about Python.

Variables and Values

The values stored in variables are pointers (addresses).

- That's why Python variables are untyped.
- That's why it's OK to assign a value of "another type" to `x`.

If you later re-assign `x`:

```
x = 2.75
```

the value 2.75 is stored in another memory location; its address is placed in the memory location denoted by `x` and the memory occupied by the value 1 may be reclaimed by the *garbage collector*.

Type Examples

```
>>> x = 1
>>> type(x)
<class 'int'>
>>> x = "abc"
>>> type(x)
<class 'str'>
>>> print (type(2.75))
<class 'float'>
>>> import math
>>> print (type (math))
<class 'module'>
```

Types in Python

Is it correct to say that there are no types in Python?

Yes and no. It is best to say that Python is "dynamically typed." Variables in Python are untyped, but values have associated types which are actually classes. In some cases, you can convert one type to another.

Most programming languages assign types to both variables and values. This has its advantages and disadvantages. Can you guess what the advantages are?

Mutable and Immutable

A *mutable* type contains values that can be changed by the program. An *immutable* type cannot be changed.

For example, the Python string type **str** is immutable. You can extract parts of a string, append it to another string, and perform various operations on it. *But you can't change it.* There are no operations that allow you remove the first character or swap one character for another in a string.

If you want to do that, you create a new string.

```
new_string = old_string[1:]
```

A list is an example of a mutable type; you can remove an item and still have the "same" list.

Mutable vs. Immutable

Consider a list (mutable) versus a tuple (immutable).

Types in Python

Type	Description	Syntax example
str	An immutable sequence of characters. In Python 3 strings are Unicode by default.	'Wikipedia' "Wikipedia" """Spanning multiple lines"""
bytes	An immutable sequence of bytes	b'Some ASCII' b"Some ASCII"
int	An immutable fixed precision number of unlimited magnitude	42
float	An immutable floating point number (system-defined precision)	3.1415927
complex	An immutable complex number with real number and imaginary parts	3+2.7j
bool	An immutable truth value	True, False
tuple	Immutable, can contain mixed types	(4.0, 'string', True)
list	Mutable, can contain mixed types	[4.0, 'string', True]
set	Mutable, unordered, no duplicates	{4.0, 'string', True}
dict	A mutable group of key and value pairs	{'key1': 1.0, 3: False}

Operators: Arithmetic Operators

These are the arithmetic operators that operate on numbers (integers or floats). The result of applying an arithmetic operator is a number.

- Addition: $+$
- Subtraction: $-$
- Multiplication: $*$
- Division: / (use // for integer division)
- Remainder: %
- Exponentiation: $**$

Relational Operators

Python has 6 comparison operators. The result of applying the comparison operators is a Boolean: `True` or `False`.

- Equal to: `==`
- Not equal to: `!=`
- Greater than: `>`
- Greater than or equal to: `>=`
- Less than: `<`
- Less than or equal to: `<=`

You may apply the comparison operator between two operands, e.g., `(a > b)`, or in sequence like so `(a > b > c)`. But this is not recommended.

Boolean Operators

Python has two Boolean literals: `True` and `False`. But Python will also regard any of the following as `False` in a Boolean context:

- the number zero (0),
- the empty string (""),
- the reserved word `None`.

All other values are interpreted as `True`.

What does the following code do?

```
x = 5
while x:
    print (x)
    x -= 1
```

Boolean Operators

There are 3 Boolean operators:

- `not`: unary operator that returns `True` if the operand is `False` and vice versa.
- `x and y`: if x is False, then that value is returned; otherwise y is evaluated and the resulting value is returned.
- `x or y`: if x is true, then that value is returned; otherwise y is evaluated and the resulting value is returned.

Returning a value before all arguments are evaluated is called *short circuit evaluation* and can be very useful.

```
if ( y != 0 and x / y > 0 ):
   print ( x / y )
```

Assignments

A variable can be assigned the value of a literal or expression.

```
a = 1                   # 1 is an integer literal
b = 2.3                 # 2.3 is a floating point literal
c = False               # False is a boolean literal
d = 5L                  # 5L is a long integer literal
e = 8 + 6j              # 8 + 6j  is a complex literal
f = "Hello"             # "Hello" is a string literal
```

Recall what actually goes into the variable! What do you think happens in each of these cases?

An expression is composed of variables and operators. An expression is evaluated before its value is used. Python allows for multiple assignments at once.

```
x, y = y, x
```

Conditionals

In Python, a conditional allows you to make a decision. You can use the keyword `elif` that is a contraction of the words `else` and `if`.

```
if (cond1):
   ...
elif (cond2):
   ...
else:
   ...
```

The body of `if`, `elif`, and `else` parts need to be indented.

Conditional Example

```
def letterGrade(score):
    if score >= 90:
        letter = 'A'
    elif score >= 80:
        letter = 'B'
    elif score >= 70:
        letter = 'C'
    elif score >= 60:
        letter = 'D'
    else:
        letter = 'F'
    return letter
```

Loops

The loop constructs of Python allow you to repeat a body of code several times. There are two types of loops: *definite* loops and *indefinite* loops.

You use a definite loop (**for** loop) when you know a priori how many times you will be executing the body of the loop.

You use an indefinite loop (**while** loop) when you do not know a priori how many times you will be executing the body of the loop.

To write an efficient loop structure you must ask yourself these questions:

- Where do I start?
- When do I end?
- How do I go from one iteration of the loop to the next?

While Loops

Syntactically the simplest loop construct is the `while` loop.

```
while (cond):
  ...
  loop_body
  ...
```

What is the semantics of this construct? I.e., what does this really do?

While Example

```
def count_down(n):
    if n < 0:
        print('You're trying to fool me!')
        return
    while n > 0:
        print (n)
        n -= 1

def ask_ok(prompt, retries=4, complaint='Yes or no, please!'):
    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:
            print("What a loser!")
            break
        print complaint
```

Range

When defining loops, a useful construct is `range`. You can think of `range(10)` as returning the list `[0, 1, 2, ..., 9]`. That's what it does in Python 2. In Python 3, it creates an *iterator*, a function to return successive values in that list as needed.

```
>>> range(10)
range(0, 10)
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(1, 11))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> list(range(0, 30, 5))
[0, 5, 10, 15, 20, 25]
>>> list(range(0, 10, 3))
[0, 3, 6, 9]
>>> list(range(0, -10, -1))
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
>>> list(range(0))
[]
>>> list(range(1, 0))
[]
```

For Loops

A definite loop requires a *loop counter* which takes on a series of values. Example: write a loop that executes the body 10 times.

```
for i in range (10):
   ...
   loop_body
   ...
```

The loop counter `i` goes through the values 0 through 9 as the loop iterates.

We can also specify the beginning and ending value of `i`.

```
for i in range (3, 15):
   ...
   loop_body
   ...
```

What values does i take on in this loop?

For Loops

We can also specify the step size for the loop counter.

```
for i in range (1, 10, 2):
   ...
   loop_body
   ...
```

What values will i take on here? How would you process the sequence backwards?

You can also enumerate a *sequence* of values for the loop counter.

```
for i in [7, 4, 18, 12]:
   ...
   loop_body
   ...
```

For Example

What does the following code do?

```
def printStringExploded(symbolString):
    for c in symbolString:
        print(c, end=" ")
    print()

def compressExplodedString(explodedString):
    for c in explodedString:
        if c != " ":
            print(c, end="")
    print()

printStringExploded("abcdefg")
compressExplodedString("a b c d e f g")
```

Functions

A function is an encapsulation of some functionality. Functions can be called at the top level or from within another function (or itself). A function may or may not return a value or values.

The structure of a function definition is as follows:

```
def function-name ([formal-parameters]):
    ...
    body-of-the-function
    ...
```

The `function-name` is an identifier. You can have zero or more *formal parameters*, but must have the parentheses.

Functions

The formal parameters are placeholders that accept values sent to it from the calling function. When a function is called, the parameters that are passed to it are called *actual parameters*.

For a complete list of built-in functions see:
`docs.python.org/library/functions.html`.

Function Examples

```
def addTwo (a, b):
  return a + b

def divide (a, b):
  return a / b, a % b

def isEven (x):
  return (x % 2 == 0)

def gcd (m, n):
  while (m != n):
    if (m > n):
      m = m - n
    else:
      n = n - m
  return m
```

Function Examples (2)

```
def coPrime (a, b):
  if (gcd(a, b) != 1):
    return
  else:
    print "%0d and %0d are co-prime" % (a, b)

def main():
  x = 2
  y = 3

  z = addTwo (x, y)

  p, q = divide (x, y)

  if (isEven(z)):
    print z

  coPrime (x, y)

main()
```

What does this program do?

User-Defined Functions

To call a user defined function that is in the same program you simply call it by name without using the `dot` operator.

If the function does not return a value, then call it on a line by itself. If it returns values then assign those return values to variables.

Import Modules

There are other functions that reside in modules that have to be loaded before you can use them: `string`, `math`, `random`, etc. You load these modules by using the `import` directive.

```
import string
import math, random
```

After you load the module you can call on the functions using the `dot` operator.

```
x = 7
y = math.sqrt (x)
z = random.random()
```

Lists in Python

The analogue of an array in Python is a `list`. A list is *heterogenous* and *dynamic*. What does that mean? The size is not specified at creation and can grow and shrink as needed.

Python provides built-in functions to manipulate a list and its contents.
There are several ways in which to create a list:

- enumerate all the elements
- create an empty list and then append or insert items into the list.

Lists in Python

```
# Enumerate the items
a = [1, 2, 3]

# Create an empty list and append or insert
a = []
a.append(1)                 #  a = [1]
a.append(2)                 #  a = [1, 2]
a.insert(1, 3)              #  a = [1, 3, 2]

# Create a two dimensional list
b = [ [1, 2, 3], [4, 5, 6], [7, 8, 9] ]
```

Lists in Python (3)

To obtain the length of a list you can use the `len()` function.

```
a = [1, 2, 3]
length = len (a)          # length = 3
```

The items in a list are indexed starting at 0 and ending at index `length - 1`.

You can also slice a list by specifying the starting index and the ending index and Python will return to you a sub-list from the starting index and upto but not including the end index.

```
a = [1, 9, 2, 8, 3, 7, 4, 6, 5]
b = a[2:5]                  # b = [2, 8, 3]
```

Lists in Python (4)

One of the most important operations that you can do with a list is to *traverse* it, i.e. visit each element in the list in order.

There are several ways in which to do so:

```
a = [9, 2, 6, 4, 7]
for item in a:
  print (item, end = " ")          # 9 2 6 4 7
```

Other useful functions to sort, reverse, etc. lists are here:
`docs.python.org/release/3.1.3/tutorial/datastructures.html`

Input and Output

Python 2 has input and raw_input functions. Python 3 has only input, which works like raw_input in Python 2. That is, it always returns a string value.

You can prompt the user to enter a string using the function `input()`. The function then waits for the user to enter the value and reads what has been typed only after the user has typed the `Enter` or `Return` key.

```
n = input ("Enter a number: ")
```

Input and Output

The `print` command allows you to print out the value of a variable. If you want to add *text* to the output, then that text has to be in single or double quotes. Print takes an arbitrary number of arguments separated by commas, and prints them separated by spaces.

```
print ('n = ', n, end=" ")
print ("name = ", name)
```

Assuming `n` contains the string "abc", what does the above print?

Input and Output

An optional `end` argument allows specifying how to finish the print. The following are equivalent:

```
print ('n = ', n, end="\n")
print ('n = ', n)
```

To suppress the newline, provide an explicit alternative.

```
print("1"); print ("2")
1
2
print("1", "2")
1 2
>>> print("1", end=""); print("2")
12
```