Review Sheet for Midterm #1 COMPSCI 119

Professor William T. Verts

Simple Data Types

There are a number of data types that are considered "primitive" in that they contain only a single value. These data types are `int`, `long`, `float`, `complex`, and `bool`.

`int` Integers (numbers without fractions) that fit into the hardware that the processor can operate on directly at full speed. Usually either 32 bit or 64 bit, which constrains the range of legal values it can support. In JES, the `int` is 32 bit, which maps onto the range from -2147483648 to +2147483647 (any integer calculation which exceeds these limits automatically defaults over to the `long` data type).

`long` Integers that are handled in software, and may have any value (including those covered by the `int` data type), but at the cost of a loss of performance. In JES, values in the `long` data type are indicated by an `L` suffix. Absorbs `int`. For example, `5+6L` is `11L` (the `5` is cast as `5L` before the addition).

`float` Numbers with fractions. The legal range of values is approximately 10^{-324} to 10^{+308}, with about 15 to 16 significant figures of precision. Calculations which exceed 10^{+308} are converted to `inf` (infinity), and calculations which are smaller than 10^{-324} go to zero. It is usually not a good idea to compare a numeric value to a float because of round-off error (even when numbers are mathematically supposed to be equal, they rarely are in practice). Absorbs both `int` and `long`. For example, `5.0+3+2L` is `10.0` (both the `3` and the `2L` are cast as floats before the addition).

`complex` Numbers with a real part and an imaginary part (both floats). Contrary to math operations that use `i` to indicate the imaginary part, Python uses `j` instead. A number which is `5` along the real axis and `7` along the imaginary axis is written in Python as `(5+7j)`. Adding an `int`, `long`, or `float` to a `complex` adds the value to the real part. For example, `(5+7j)+10` is `(15+7j)`.

`bool` Short for Boolean (named after George Boole). Only has two possible values, `False` and `True`. The result of any comparison is always a `bool`. For example, `5<7` is `True`, but `5>7` is `False`. In a weird type violation, `False` is equivalent to `0` and `True` is equivalent to `1`, so an expression such as `9+(5<7)` is the same as `9+True`, which equals `10` as the result.

Variables

Variable names can be of any length, but must start with a letter, may contain letters or digits or the underscore. Upper case is distinct from lower case. It is almost never a good idea to have two variables with the same letters that differ only by case (`FROG` and `frog`, for example).

Variables can appear on either side of the equal sign in an assignment statement, but expressions can only appear on the right side. For example, the expression `N = N + 1` is legal, but `N + 1 = N` is illegal. In a statement such as `N = N + 1`, the old value of `N` is used in the calculation, and then the result becomes the new value of `N`.

Variables are dynamically typed, which means that they take on the data type of whatever they are assigned. For example, the statement `N = 5` makes `N` an `int`, but if that statement is followed by `N = 4.5` then `N` becomes a `float`.

Basic Structures

In Python statements are normally written one per line, and program execution goes from one statement to the next in strict linear sequence. All statements must have the same level of indentation. Statements that enclose other code such as the `if` and the `while` statement indent the statements that they enclose.

To make a selection, use the `if` statement:

```
if (condition):
    # do something here
elif (condition):
    # do something here
elif (condition):
    # do something here
elif (condition):
    # do something here
else:
    # do something here
```

In a structure such as this, there need not be any `elif` or `else` clauses. The following are both legal:

```
if (condition):
    # do something here
else:
    # do something here
```

```
if (condition):
    # do something here
```

To do something more than once, use a `while` loop:

```
while (condition):
    # do something here
```

Note that `while` loops come in two "flavors". One is open-ended in the number of times that the loop body executes, and the other is more fixed. There is no guarantee that the following loop will execute any particular number of times because it is dependent on user input:

```
X = input("Enter a positive number")
while (X < 0):
    X = input("Enter a positive number")
```

but the following loop will execute exactly `N` times (for any positive value of `N`):

```
N = input("Enter a positive number")
Counter = 0
while (Counter < N):
    # do something here for each value of Counter 0,1,2,…,N
    Counter = Counter + 1
```

Functions and Parameter Passing

Functions have a meaning similar to that of mathematics. For example, the square root function `sqrt` receives a value through its argument list and returns a value through its name. Even though it is already predefined (as part of the math library, so we would write `math.sqrt`), if we were to write it ourselves we would set up a template such as:

```
def sqrt(N):
    # Compute the result
    # and put the value into
    # variable Result
    return Result
```

Note that the body of the function is indented relative to the `def` statement. This function has one parameter, called `N`, which is local to the function. That is, it is a socket into which a value is plugged from outside when the function is called, but the value of `N` vanishes when the function is complete. There can be other variables named `N` elsewhere, but they have nothing to do with this `N`. Calls to this function can be any of the following forms:

```
X = sqrt(2)
X = sqrt(N)    # Not the same N!
X = sqrt(input("Enter a positive number"))
```

Not all functions return a value. If they do something but don't need to pass back a value, then the `return` statement will stand by itself. For example, a function that does nothing but print, or call other functions, or the main program, would look something like this:

```
def Main():
    N = input("Enter a positive number")
    X = sqrt(N)
    print "The square root is", X
    return
```

There may be as many parameters to a function as needed, but the call to that function must supply a value for each parameter, unless there are default parameters. For example, the following function may be called with 2, 3, or 4 parameters (default parameters may only appear at the right end of the parameter list):

```
def MyFunction(A,B,C=0,D=1):
    # do something here
    return
```

Example calls

```
MyFunction(1,2,3,4)      →   A=1, B=2, C=3, D=4
MyFunction(1,2,3)        →   A=1, B=2, C=3, D=1
MyFunction(1,2)          →   A=1, B=2, C=0, D=1
MyFunction(1)            →   Error (not enough parameters)
```

The `return` statement is generally placed at the end of a function, but it may also appear elsewhere. For example, the following function has the `return` statement in two different places:

```
def MyFunction(N):
    if (N < 0):
        return "Error"
    X = math.sqrt(N)
    return X
```

Lists, Tuples, and Strings

Lists, tuples, and strings are all structured data types that can contain 0 or more individual values. Tuples and strings are immutable in that once defined you cannot change the length or individual values within them (with few exceptions). Lists are mutable, and their contents can be changed after the fact. Lists and tuples can contain any data type; strings are essentially lists of characters. Each has their own constructors:

Lists	`[value,value,value,…,value]`	
Tuples	`(value,value,value,…,value)`	
Strings	`"xxxxxxxxxxxxxxxxxx"`	where all `x` are characters.
or	`'xxxxxxxxxxxxxxxxxx'`	
or	`"""xxxxxxxxxxxxxxxxxx"""`	string may span lines.
or	`'''xxxxxxxxxxxxxxxxxx'''`	string may span lines.

These items may all be empty. Empty lists are `[]`, empty tuples are `()`, and empty strings are either `""` or `''`.

The `len` function will tell you how many items are in any of these structures.

Individual items are indexed by numbers from 0 up to the length minus 1. For example, in the list `L = [1, 6, 9, 2, 7]`, the `1` is at `L[0]` and the `7` is at `L[4]`. `L[5]` does not exist. However, negative indexes are allowed: `L[-1]` is the same as `L[5]` in this list. More generally, `L[-1]` is the same as `L[len(L)-1]` and `L[0]` is the same as `L[-len(L)]`.

Tuples and strings work the same way, and also use brackets to indicate indexes. For example, in the string `S = "Frog"`, `S[0]` is `"F"` and `S[-1]` is `"g"`.

To illustrate mutability versus immutability, `L[3] = 4.5` is legal because lists are mutable, but the statement `S = "Q"` is illegal because strings are immutable.

To add something to either, you can append a value to either end, and then assign the new value back to the original variable:

```
L = L + [10]    # append 10 to the right end of the list
L = [10] + L    # append 10 to the left end of the list
S = S + "Q"     # append Q to the right end of the string
S = "Q" + S     # append Q to the left end of the string
```

The range function and the for loop

There is another type of loop in addition to the while loop. The for loop is always of the form:

```
for variable in list:
    # do something
```

The variable takes on each value from the list as it goes through the body of the loop. For example, the following loop:

```
for X in [4,1,7,2]:
    print X
```

will print:

```
4
1
7
2
```

This also works for strings, which are effectively lists:

```
for X in "Frog":
    print X
```

will print:

```
F
r
o
g
```

The important thing about for loops is the list of values it steps through. Generating the correct lists is therefore a critical task. This is the job of the `range` function:

`range(N)`	generates	`[0,1,2,3,…,N-1]`
`range(M,N)`	generates	`[M,M+1,M+2,M+3,…,N-1]`
`range(M,N,P)`	generates	`[M,M+P,M+2P,M+3P,…,N-1]`

For example:

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

The `range` function works well in concert with the `for` loop:

```
for X in range(4):
    print X
```

will print:

```
0
1
2
3
```

The `range` function also works well in concert with lists or strings:

```
L = [2,8,1,4,5]
for X in range(len(L)):
    print L[X]
```

While `X` will take on successive values `0`, `1`, `2`, `3`, `4`, those values are indexes into `L`, so the code will actually print:

```
2
8
1
4
5
```

Exception handling

Sometimes it is possible to check for all possible error conditions before a computation is allowed to proceed:

```
if (N > 0):
    Result = M / N
else:
    Result = 0.0
```

This isn't always possible, so the alternative is to proceed with the calculation as if nothing was wrong, but pick up problems that would crash the program only when and if they occur.

```
try:
    Result = M / N
except:
    Result = 0.0
```

Other items

The body of an `if`, `elif`, `else`, `while`, or `for` cannot be empty, but in cases where there is nothing to do there the `pass` statement can be inserted to act as a placeholder:

```
if (condition):                while (condition):
    pass                           pass
```

If the body of an `if`, `elif`, `else`, `while`, or `for` consists of a single statement, then that statement may be placed on the same line:

```
for X in range(10):        →    for X in range(10): print X
    print X
```

You cannot do this if the body contains more than one statement.