The current topic: Python

- ✓ Introduction
- Object-oriented programming: Python
  - ✓ Features, variables, numbers, strings, Booleans, while loops
  - – Next up: If statements, sequences, functions, modules
- Types and values
- Syntax and semantics
- Functional programming: Scheme
- Exceptions
- Logic programming: Prolog

Announcements

- Tutorials start today in SF1012.
- Lab 1 should be posted by Monday.
  - – Due: September 29th, 10:30am
- Term tests:
  - – October 6th, 11:10-12:00, **GB405**
  - – November 3rd, 11:10-12:00, **GB405**

Problems with the C/Java If statement

- Consider the following C/Java code:

```
mark = 85;
comment = "pass";
if (mark >= 50)
    if (mark >= 90)
        comment = "excellent";
else
    comment = "fail";
```

- What is the final value of `comment`?
  - – `"fail"`!
  - – The `else` actually belongs to the inner `if` statement.
  - – The indentation suggests a meaning to us, but is ignored by the compiler.
  - – Python's approach to indentation solves this problem. (And prevents bugs!)
- Reference: Sebesta, Chapter 8.

Conditional expressions

- Used to write simple if/else statements in a single line. Instead of:

```
if mark >= 50:
    grade="pass"
else:
    grade="fail"
```

we can write:

```
grade = "pass" if mark >= 50 else "fail"
```

- New in Python 2.5, and similar to the ? operator in C/Java.
  - – In C/Java, the statement above would be written:

```
grade = (mark >= 50) ? "pass" : "fail"
```

  - – Which form is more readable?

Containers and sequences

- A *container* is a collection of objects.
  - In Python, an object can be practically anything.
  - Properties of a container include:
    - Size
    - Whether or not a particular object is contained
    - Whether or not objects can be added or removed
- A *sequence* is a container with some kind of ordering.
  - A sequence can contain a particular object more than once.
  - Sequences can be indexed to obtain the *N*-th object in the sequence.
- In Python:
  - Sequence types are *lists*, *tuples*, and *strings*.
  - Non-sequence container type is the *dictionary*.
- Reference: Sebesta, Chapter 6

Lists

- A *mutable* sequence of objects.
  - Mutable: Lists can be modified (objects can be added and removed).
- Lists can be *heterogeneous.*
  - Can have different kinds of objects in the same list.
- Similar to arrays and vectors in C++/Java.
- Simple examples:

```
ls = [1, "CSC", 2, [], "321"]
ls[0]   # 1
ls[3]   # []
ls[5]   # IndexError
```


Lists

- More examples:

```
ls = [1, "CSC", 2, [], "321"]
i = 0
while i < len(ls):  #len returns size of the list
    ls[i] = i
    i+= 1
ls    # [0, 1, 2, 3, 4]
```


Nested lists

- Nested lists:

```
nest = [[1, 2], [3, 4], [5, 6]]
nest[0]       # [1, 2]
nest[0][1]    # 2
y = nest[0]
y[1] = 4
nest[0][1]    # 4  (and not 2!)
y = [99, 99]
nest[0][1]    # 4  (and not 99!)
```

- Observe that values associated with Python variables are actually references! This should remind you of pointers in C or references in Java.
  - In the example above:
    - `nest` is a reference to a 3-element list
    - `nest[0]` is a reference to the first element of `nest`
    - `y=nest[0]` sets `y` to be a reference to the object that `nest[0]` refers to.
    - `y[1]=4` modifies the object that `y` refers to.

Adding to lists

- Use the append method to add to a list.

```
ls = [32]
ls.append(4)
ls    # [32,4]
# Do not do the following:
ls = ls.append(5)      # ls is now None

ls2 = ["cat", "dog"]
ls2.append(["rabbit"])
ls2   # ["cat", "dog", ["rabbit"]]
```


Removing from lists

- Use the pop method to remove and return the last element of a list.

```
ls = [32, 1, [], "fish", "mouse"]
last = ls.pop()
last   # "mouse"
ls     # [32, 1, [], "fish"]
```

- Observe that using append and pop, we can treat a list as a stack.
- What if we want to remove an element other than the last one? Pass an index to pop.

```
ls = [32, 1, [], "fish", "mouse"]
last = ls.pop(3)
last   # "fish"
ls     # [32, 1, [], "mouse"]
```


Adding lists together

- We can create a new list that is the concatenation of two existing lists.

```
ls1 = ['a', 'e', 'i', 'o', 'u']
ls2 = ['b', 'c', 'd', 'f']
ls3 = ls1 + ls2
ls3   # ['a', 'e', 'i', 'o', 'u', 'b', 'c', 'd', 'f']
```

- We can create a new list than consists of multiple copies of an existing list.

```
ls = [[5, 10]]
ls2 = ls*3
ls2            # [[5, 10], [5, 10], [5, 10]]
ls2[0][0]=1
ls2            # [[1, 10], [1, 10], [1, 10]]
ls2[0] = 'w'   # ['w', [1, 10], [1, 10]]
```

- Remember that lists store references, so when we copy a list we get a copy of the stored references.

Strings

- Strings are essentially lists of characters except:
  - Strings are immutable. You cannot change an existing string object. Instead, you have to create a new object.
  - There isn't a "character" type in Python.
  - But, intuitively, you can think of strings as unchangeable lists where each element is a single character.
  - As we saw last time, this is exactly what happens when we use a string as a list.

```
s = 'bear'
s[2]      # 'a'
t = ['teddy']
t += s
t         # ['teddy', 'b', 'e', 'a', 'r']
```

Tuples

- Tuples are immutable lists. You cannot add or remove elements from a tuple.
- Normally enclosed in round brackets, but this isn't always necessary.

```
t1 = (2, 3, 4)
t1[2]       # 4
t2 = 2, 3, 4
t1 == t2    # True
t3 = (1)    # t3 is *not* a tuple, it's the int 1
t4 = (1,)   # t4 is a tuple with one element
t5 = 1,
t4 == t5    # True
```

- Observe that tuples of size one need a trailing comma to differentiate them from a single object.

Swapping values

- Tuples make it easy to swap values.

```
a = 5
b = 6
a, b = b, a
a, b       # (6, 5)
```

- Python creates a new tuple (in the background) to store the right side when it evaluates `a,b = b,a`, and then it assigns from this new tuple to the variables on the left.

Slicing

- Slicing allows us to extract a portion of a sequence (rather than just a single element).

```
ls = [5, 6, 7, 8]
ls[1:4]      # [6, 7, 8]
ls[1:3]      # [6, 7]
ls[2:]       # [7, 8]
ls[:2]       # [5, 6]
s = "hello"
s[1:4]       # "ell"
t = (5, 6, 7, 8)
t[0:2]       # (5, 6)
```

- A slice is of the form:
  `<sequence>'['<start-index>':'<end-index + 1>']'`
- When the start index is omitted, the slice starts at the beginning of the sequence.
- When the end index is omitted, the slice ends at the end of the sequence.

Negative and out-of-range indices

- Negative indices count backward from the end of a sequence. The last item in a sequence has index -1.

```
ls = ["c", "e", "f", "d", "g"]
ls[-1]    # "g"
ls[-2]    # "d"
ls[1:-1]  # ["e", "f", "d"]
ls[1:-2]  # ["e", "f"]
```

- Out-of-range indices cause errors in single-element references but not in slices.

```
ls = ["c", "e", "f", "d", "g"]
ls[99]    # IndexError
ls[3:99]  # ["d", "g"]
ls[-10:-8]  # []
```

Splicing

- *Splicing* is assigning to a slice.
  - This only works for mutable sequences, since it modifies the sequence.
    - So it works for lists, but not for strings or tuples.
- The list on the right-hand side replaces the slice being assigned to.
  - The list on the right-hand side does **not** have to be the same length as the slice being assigned to.

```
ls = [5, 6, 7, 8]
ls[1:2] = [11, 12, 13]
ls     # [5, 11, 12, 13, 7, 8]
ls[1:2] = 9     # TypeError
ls[1:3] = []
ls     # [5, 13, 7, 8]

s = 'hello'
s[1:3] = 'abc'     # TypeError, since strings are immutable.
s = s[:1] + 'abc' + s[3:]    # Creating a new string works.
s      # 'habclo'
```


For loops

- Look a little like for loops in C/Java, but the idea is different.

```
for c in ['ab', 'cd', 'ef']:
    print c*2,
#output is: abab cdcd efef

for d in 'friday':
    print d,
#output is: f r i d a y
```

- The general form is:
  `'for' <variable> 'in' <container>':'`
  - Which means "do the following for each member of this container".
  - If the container is a sequence, the members are processed in order.

Counting

- The `range()` function makes a list of numbers.

```
range(3)            # [0, 1, 2]
range(2, 5)         # [2, 3, 4]
range(2, 10, 3)     # [2, 5, 8]
range(3, 1)         # []
range(3, 1, -1)     # [3, 2]


s = 'python'
for i in range(len(s)):
    print i, s[i],
#output is: 0 p 1 y 2 t 3 h 4 o 5 n
```


Some little things

- In while loops and for loops, `break` and `continue` behave the same as in C/Java.
  - `break` exits the innermost loop
  - `continue` moves on to the next iteration of the innermost loop
- The operator `in`:
  - `x in c`: This is `True` if and only if the value `x` is in the container `c`.

```
vowels = 'aeiou'
for v in vowels:
    if v in 'spaghetti':
        print v,
#output is: a e i
```

Functions

- Defined using the keyword `def`.

```
def sumOfThree(x, y, z):
    return x+y+z

sumOfThree(3, 4, 5)         # 12
sumOfThree("a", "b", "c")   # "abc"
```

- Observe that the argument types and return type are not declared.
- Any assignments within a function are treated as assignments to local variables (that is, unless a `global` statement is used).

Functions: Passing arguments

- Arguments are passed by copying references (not by copying objects).

```
def changeMyArgs(x, L):
    x = 12
    L[0] = 32

a = 0
ls = [1, 2, 3]
changeMyArgs(a, ls)
a   # still 0, not 12
ls  # [32, 2, 3]
```


Modules

- Any Python file can be loaded as a *module* using `import`.

```
import xyz    #loads contents of xyz.py as module xyz
```

- During loading:
  - The code in the module is executed.
  - That is:
    - Classes, functions, variables are defined.
    - Executable statements are executed.
- Naming:
  - Item `x` from module `m` is referred to as `m.x`.

Modules: Comparison with Java

- In Java, `import m` doesn't "do" anything, in the sense that it generates no code and causes no action. It just allows you to refer to `m.x` as simply `x`.
- In Python, `import m` causes nearly the same actions as running `m.py`, and you still have to say `m.x` to refer to item `x` from `m`.
  - To get short-form naming in Python, use
    `from m import x`
    which imports `m` (if it hasn't been imported already) *and* allows to refer to `m.x` as simply `x`.
    - By itself, this doesn't let you refer to anything else from `m`, not even using long-form naming. But you can still do a standard import to get this, and doing so won't re-execute `m`.
  - To get short-form naming for everything in `m`, use
    `from m import *`
    but this is viewed as a somewhat dangerous practice. (Why?)

Modules: Who's in charge?

- A module's `__name__` attribute tells you whether it's the module that was called to start the program.
  - If `__name__` is `'__main__'`, the module is being run directly.
  - Otherwise, `__name__` is the module's actual name, and some other module imported it.
- This allows a module to behave differently depending on how it's being run:

```
if __name__ == '_ main__':
    #do stuff
else:
    #do other stuff
```


Modules: Where does Python find them?

- There are standard locations where Python looks for modules when it encounters an `import` statement.
- This list of locations is stored in `sys.path`. To access it:

```
import sys
sys.path    # This is the list of locations.
```

- `sys.path` is just an ordinary list, so you can change it if you need to.

Exercises

- Write a function that takes an integer n as a parameter and returns a list containing the first n Fibonacci numbers.
- Put your function in a file, and use this file as a module. Try importing the module with and without short-form naming.
- Challenge: Try writing your function without any loops. (Hint: Use recursion. This is a preview of functional programming.)