Intro to Computing ES112

Lecture 2

Chap 8,9,10 from ThinkPython book

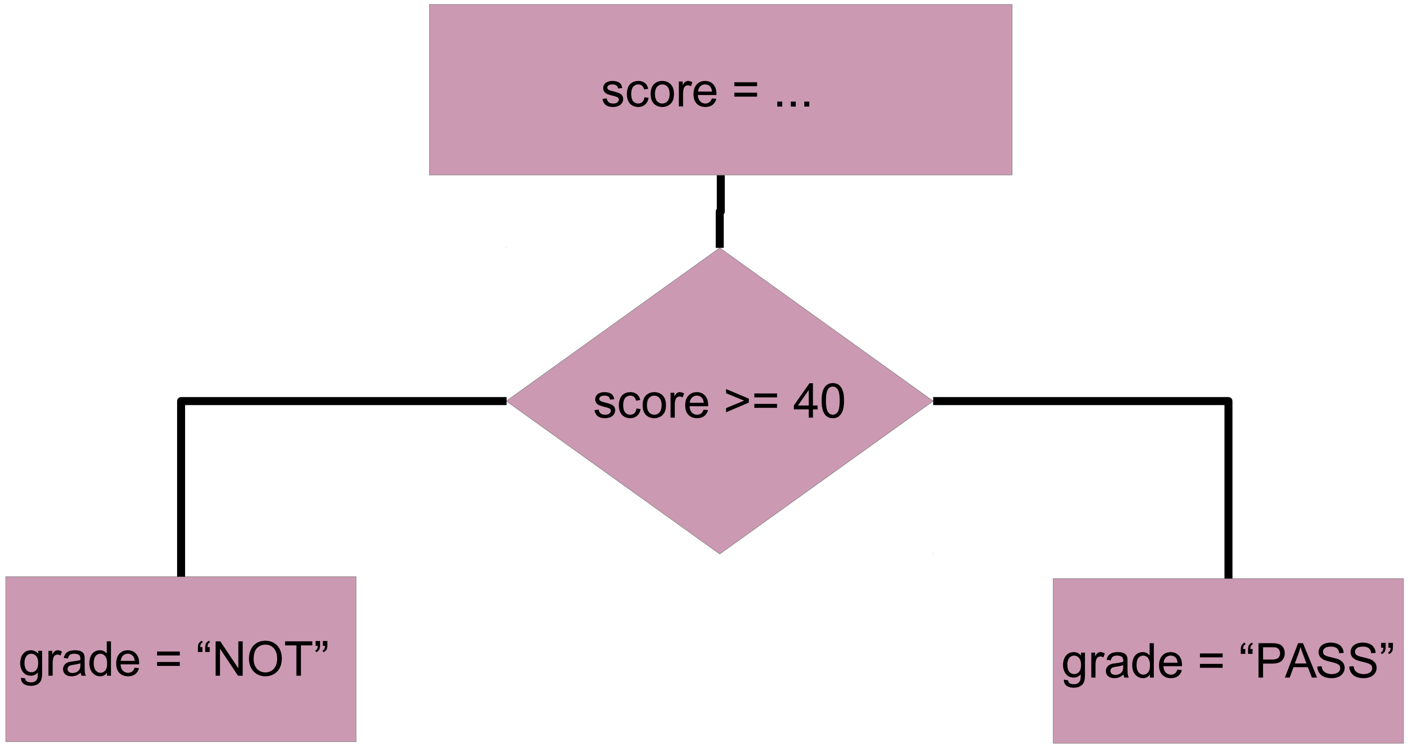
Conditionals + Loops

In all the examples we have seen till now, we always execute all the python statements

What if we want to execute some statements some of the time, based on some condition?

Branching programs

- Programs are not very useful unless they contain some logical decisions



if...else

- Suppose we want to have a problem like this in determining a student's grade
  - If testscore > 40, then grade = PASS
  - else not ….
- In python

```
if score >= 40:
   grade = “A”
else:
   grade = “B”
```

Note the indentation and the colons

Indentation

- The sole most important thing when programming in python

- Indentation determines the program flow

```
if score >= 40:
    grade = "A"
else:
    grade = "B"
```

```
if score >= 40:
    grade = "A"
else:
grade = "B"
```

IndentationError: unindent does not match any outer indentation level

Indentation

Use a single tab (or fixed number of spaces) as indentation for each level

Be careful of following types of mistakes

```
if score >= 40:
   grade = “A”
   agrades = agrades + 1
else:
   grade = “B”
   bgrades = bgrades + 1
```

```
if score >= 40:
   grade = “A”
   agrades = agrades +
else:
   grade = “B”
bgrades = bgrades + 1
```

Indentation

Use a single tab (or fixed number of spaces) as indentation for each level

Be careful of following types of mistakes

```
if score >= 40:
   grade = "A"
   agrades = agrades+1
else:
   grade = "B"
   bgrades = bgrades+1
```

```
if score >= 40:
   grade = "A"
   agrades = agrades+1
else:
   grade = "B"
bgrades = bgrades+1
```

Forgetting to indent properly will cause logical errors in your program, which are hard to catch

if...else

- Boolean expression = results in `True` or `False`
  - This is of type `bool`
  - Examples: `1 < 2,  x! = 1,  x == y`
- The expression following `if` keyword should be a boolean expression

```
grade = “Y”
if score >= 40:
   grade = “PASS”
else:
   grade = “NOT”
```

boolean

Chained conditionals if....elif...else

- What if we have more conditions? e.g. assign grades

```
grade = “A”
if score >= 90:
   grade = “A”
elif score >= 80:
   grade = “B”
elif score >= 70:
   grade = “C”
else:
   grade = “D”
```

The middle conditions are all `elif`

Nested conditionals

- What if we have more complicated conditions

```
grade = “A”
if testscore >= 90:
   if labscore >= 90:
      grade = “A+”
   else:
      grade = “A”
elif testscore >= 80:
   if labscore >= 90
      grade = “B+”
   else:
      grade = “B”
elif...
   ...
```

Two different scores – `testscore` and `labscore`

Note the indentation – indicates the nesting

More complicated grading

- Write down a function to determine a grade as follows:
  - `totalscore = testscore + labscore`
  - First grade is determined as “A”, “B”, “C” or “D” as per `totalscore`.
  - If `labscore > testscore`, then A becomes A+, B becomes B+ etc...

Structured Types

- Up until now we have seen the following:
  - int, float, string (somewhat)
- Structured types represent collections
- Three such main types in python
  - string
  - tuple
  - List
  - dictionary

Types

```
listexample = [1, 2, 'this']
stringexample = “this is a string”
tupleexample  = (1, 3, 'this', “”)
anothertuple = 1,3,'this'
```

- The formats are slightly different
- Try printing each of the above
- Note that the brackets when creating the tuple are optional, the comma is important

Structured types

- All the structured types are essentially sequences
- Which means that they all have a set of common methods:
  - accessing individual element : `s[i]`
  - Getting the length : `len(s)`
  - Accessing a subsequence: `s[a:b]`
- Note that they all are 0-indexed, which means that indices range from `0 to len(s) - 1`

Simple exercises

- What happens when we do each of the following:

```
fruit =  “banana”
print fruit[1]
print fruit[10]
print fruit[1.5
print fruit[-2]
print fruit[-20]
print fruit[len(fruit)]
```

- Why are negative indices are useful?
- Lets try the same with

```
fruit = ['b','a','n','a','n','a']
```

What are the differences?

	Type of elements	Mutability
List	arbitrary	Yes
String	only characters	No
Tuple	arbitrary	No

Mutability means that we can assign `s[1] = ….` after we create it.

Why do we need different types then?

What are the differences?

	Type of elements	Mutability
List	arbitrary	Yes
String	only characters	No
Tuple	arbitrary	No

Mutability means that we can assign `s[1] = ….` after we create it.

Why do we need different types then?

- Having only ascii characters in strings helps us write special functions for strings
- Non-mutability is sometimes useful, when you do not want others to change your data

String operations

- Given that fruit is a string, what does `fruit[:]` mean?
  - What is `fruit[1:]`
  - What is `fruit[:4]`

- How can we convert the string "Hello world" into the string "Jello world" ?

String methods

- A method is a function, but associated with a particular type, and the syntax for it is a little different

```
>>> word = 'banana'
>>> new_word = word.upper()
>>> print new_word
BANANA
```

- Note that there is no input to the “upper” function
  - Instead note the dot notation
- The string that it is called upon is the implicit input
- This is a feature of object-oriented nature of python

Other string methods

```
ex = “&&&&This is some string%%%%“
ex.capitalize()
ex.strip()

ex.find('s')
ex.find('s', 6)

ex.replace('s', 'S')
ex.strip()

ex.index('s')
ex.isalpha()
ex.isnum()
ex.upper()
ex.lower()
```

- Try out these too – what do each of them do?
- Check out the complete list at: https://docs.python.org/2/library/stdtypes.html#string-methods

The “in” operator

- Suppose we want to see whether a substring is present in a string:
  - e.g. does “banana” contain the substring “nana” ?

```
>>> 'nana' in 'banana'
True
>>> 'seed' in 'banana'
False
```

String comparison

- The comparison operators work on strings

```
if word == 'banana':
    print 'All right, bananas.'
if word < 'banana':
    print 'Your word,' + word + ', comes before banana.'
elif word > 'banana':
    print 'Your word,' + word + ', comes after banana.'
```

- The order is “dictionary” ordering, but with uppercase coming before lower case etc...

Lists

- Lists are sequences of arbitrary values

```
list1 = [10, 20, 30, 40]
list2 = ['crunchy frog', 'goat bladder', 'crow vomit']
list3 = ['spam', 2.0, 5, [10, 20]]
```

- The last example contains one list inside another list
  - These are “nested lists”

Creating a list

- By assigning any list, or an empty list to a variable

```
newlist = [1, 2]
another = []
```

- Accessing a list is similar to accessing elements of a string
  - Any integer expression can be used as an index.
  - If you try to read or write an element that does not exist, you get an `IndexError`.
  - If an index has a negative value, it counts backward from the end of the list.
- How will you access the elements of the inner list in list3?

Creating a list

- By assigning any list, or an empty list to a variable

```
newlist = [1, 2]
another = []
```

- Accessing a list is similar to accessing elements of a string
  - Any integer expression can be used as an index.
  - If you try to read or write an element that does not exist, you get an `IndexError`.
  - If an index has a negative value, it counts backward from the end of the list.
- How will you access the elements of the inner list in list3?

```
list3[3][0]
```

Special ways of creating lists

- `x = range(a,b)` initializes x to be the list `[a,a+1, ….b – 1]`
  - `X = range(1, 100)`
  - `Y = range(100, 1)`
  - `Z = range(1, 1000, 2)` ??

Traversing a list

- Most common way is by using for loop

```
chaats = ['bhel', 'dahi', 'bombay']
for chaat in chaats:
    print “chaat name ” + chaat

for i in range(len(chaats)):
    print “chaat name “ + chaats[i]
```

The for loop variable takes values over list elements

The for loop variable takes values over the indices

What happens in these cases ?

```
for x in []:
    print 'To print or not to print?'
```

```
for x in ['bhel', 'dahi', ['bombay', 'delhi']]:
    print 'chaat name ', x
```

List operations

Joining two lists	`list1  + list2`
Getting a sub-list (called slice)	`list1[a:b]`
Adding one element to the end of existing list	`list1.append(elem)`
Adding a number of elements to the end of list	`list1.extend(list2)`
Sorting a list	`list1.sort()`

Removing an element

- If you know the index of the element then do `pop(index)`

```
x = list1.pop(1)
```

- If you just know the element itself then

```
list1.remove(elem)
```

  - If there are multiple copies of elem in list1?

- Another way to delete the i-th element

```
del list1[i]
```

List to string and back

- A number of convenient methods for converting a list to a string and back

String → list

```
s = 'spam'
t = list(s)
print t
```

String → list

```
s = 'pining for the fjords'
t = s.split()
print t
```

String → list using delimiter

```
s = 'spam-spam-spam'
delimiter = '-'
s.split(delimiter)
```

List → string using delimiters

```
t = ['pining', 'for', 'the', 'fjords'
delimiter = ' '
print delimiter.join(t)
```

Special ways of creating lists

- `x = range(a,b)` initializes x to be the list `[a,a+1, ….b – 1]`
  - `X = range(1, 100)`
  - `Y = range(100, 1)  ??`
  - `Z = range(1, 100, 2) ??`
- `range(a, b, c)` gives `[a, a+c, a+2*c, ….b - 1]`

For loops

- Structure of a for loop will be

```
for x in range(100):
    print x
    …
print “done loop”
```

```
for x in range(100):
    y = x + y
```

```
for x in [1,2,[3,4]]:
    print x
```

```
for s in “a string”:
    print s
```

Exercise

- Write a program called `is_sorted.py` that takes a list as a input and returns True if the list is sorted in ascending order and False otherwise.

  For example, `is_sorted([1,2,2])` should return `True` and `is_sorted(['b','a'])` should return `False`

TIP: To input a list at one go, you can do the following

```
>>> x  = input()
[1, 2, 3, 4, 5]
```

Exercise

- Write a program `is_reverse.py` that takes as input two numbers `a, b` and checks whether string a is the reverse of string b

Exercise

- Write down a function that
  - Reads in a list of numbers
  - Calculate the mean and standard deviation
  - For each number x, compute (x – mean)/s.d. and print out the resulting list of numbers

Exercise

- "abecedarian" means that the letters in the string appear alphabetically
- Write a program called `is_abecedarian` that returns `True` if the letters in a string appear in alphabetical order (repeated letters are ok) and `False` otherwise.
  - The input string should only contain letters a-z or A -Z .
  - Capital and small letters should be considered same.