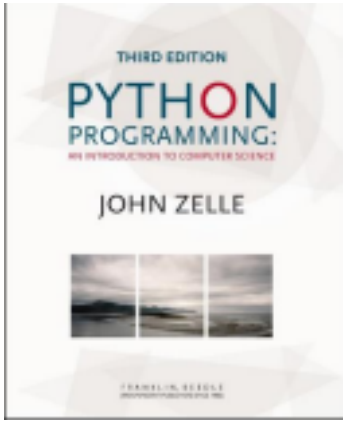
Python Programming: An Introduction to Computer Science



Seminar 2

Chapter 5 Sequences: Strings, Lists, Tuples and Files
Chapter 7 Decision Structures

Objectives

- To be familiar with various operations for sequence data types.
- To apply string formatting for program output.
- To perform basic file processing in Python.

Objectives

- To apply *simple decision*, *two-way decision, multi-way* decision
- To formulate Boolean expressions
- To implement algorithms that employ decision structures, including those that employ sequences of decisions and nested decision structures.

The Sequence Data Type

- Values of sequence data types are ordered collections of items called **elements** ■ **`str`** (immutable)
  - Elements are characters enclosed within quotation marks (") or apostrophes (') e.g., `"Ann"`
  - **`list`** (mutable)
    - Elements are values of any data type, enclosed within square brackets e.g., **`[`**`[1, 2], 'Ann', 3.3`**`]`**
  - **`tuple`** (immutable)
    - sequence of values of any data type, enclosed within round brackets e.g., **`(`**`[1, 2], 'Ann', 3.3)`

Accessing Elements via Indexing

Indexing From Left

- We can access the individual elements in a sequence through indexing.
- The positions in a sequence are numbered from the left, starting with 0.
- The general form is `<seq>[<expr>]`, where the value of expr determines which element is selected from the sequence.

H e l l B o b

0 1 2 3 4 5 6 7 8

- In a string of n characters, the last character is at position n-1 since we start counting with 0.

```
>>> greet = "Hello Bob"
>>> greet[0]
'H'
>>> print(greet[0], greet[2], greet[4])
H l o
>>> x = 8
>>> print(greet[x - 2])
B
```


Indexing From Right

H e l l o B o b

0 1 2 3 4 5 6 7 8
-9 -8 -7 -6 -5 -4 -3 -2 -1

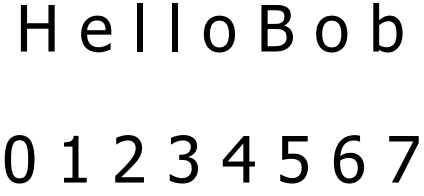
- We can index from the right side using negative indexes, starting with -1.

```
>>> greet[-1]
'b'
>>> greet[-3]
'B'
```


Accessing Contiguous Elements

Accessing Elements via Slicing

- Slicing: Accessing a contiguous sequence of elements.
- `<seq>[<start>:<end>]`
  - `start` and `end` should both be `int`
  - The slice contains
    - the elements beginning at position `start`, and
    - runs up to but doesn't include the element at position `end`.




```
<seq>[<start>:<end>]
```

If either expression is missing, then the start or the end of the sequence are used.

```
>>> greet[0:3]
'Hel'
>>> greet[5:9] ' Bob'
>>> greet[:5] 'Hello'
>>> greet[5:] ' Bob'
>>> greet[:] 'Hello Bob'
```


Combining Elements

- Concatenation "glues" two sequences together (+)
- Repetition builds up a string by multiple concatenations of a string with itself (*)

```
>>> t1 = (1, 2)
>>> t2 = (3,)
>>> t1 + t2
 (1, 2, 3)
>>> t2*5
 (3, 3, 3, 3, 3)
```


Length and Looping

- The function len will return the length of a sequence.

```
>>> len("spam")
4
```

- Iteration through elements in sequence

```
>>> for ch in "Spam!":
    print (ch, end=" ")

S p a m !
```


Summary

+	Concatenation
*	Repetition
<sequence>[]	Indexing
<sequence>[:]	Slicing
len(<sequence>)	Length
for <var> in <sequence>	Iteration through characters

Mutable vs Immutable

- Lists are mutable, meaning they can be changed.

- Strings and tuples are immutable, their values can **not** be changed.

```
>>> myList = [34, 26, 15, 10] >>> myList[2]
15
>>> myList[2] = 0
>>> myList
[34, 26, 0, 10]
>>> myString = "Hello World" >>> myString[2]
'l'
```

```
>>> myString[2] = "p"

Traceback (most recent call last):
  File "<pyshell#16>", line 1, in -toplevel-
    myString[2] = "p"
TypeError: object doesn't support item assignment
```


Useful String Functions

- `split`
  - split a string into substrings

- based on spaces.

```
>>> "Hello string methods!".split()
      ['Hello', 'string', 'methods!']
```

- based on character, supplied as a a parameter.

```
coords = input("Enter the point coordinates
x,y):").split(",")
x,y = float(coords[0]), float(coords[1])
```


More String Methods

`s.capitalize()`	Copy of s with only the first character capitalized
`s.lower()`	Copy of s with all characters in lowercase

`s.upper()` Copy of s with all characters in uppercase

`s.title()` Copy of s; first character of each word capitalized

`s.count(substr)` Count the number of occurrences of substr in s

More String Methods

`s.center(width)` Center s in a field of given width

`s.rjust(width)` Like center, but s is right-justified

`s.ljust(width)` Like center, but s is left-justified

`s.join(list)` Concatenate list of strings into one large string using s as separator.

`s.lstrip()` Copy of s with leading whitespace removed

`s.rstrip()` Copy of s with trailing whitespace removed

More String Methods

`s.count(substr)` Count the number of occurrences of substr in s

`s.find(sub)` Find the first position where sub occurs in s

`s.rfind(sub` Like find, but returns the right most position

`s.replace(oldsub, newsub)` Replace occurrences of oldsub in s with newsub

`str(expr)` Convert expr to string

String Formatting

- `<template-string>.format(<values>)`

```
"The total value of your change is ${0:0.2f}".
format(total)
```

  - `{}` : "slot" into which the value is inserted.
  - Each slot has description that includes format specifier `{0:0.2f}`

```
<width>.<precision><type>
```

<index>:<format-specifier>

String Formatting

```
>>> "Hello {0} {1}, you may have won ${2}" .format("Mr.", "Smith",
10000) 'Hello Mr. Smith, you may have won $10000'

>>> 'This int, {0:5}, was placed in a field of width
5'.format(7) 'This int, 7, was placed in a field of width 5'

>>> 'This int, {0:10}, was placed in a field of witdh
10'.format(10) 'This int, 10, was placed in a field of witdh 10'

>>> 'This float, {0:10.5}, has width 10 and precision
5.'.format(3.1415926) 'This float, 3.1416, has width 10 and precision 5.'

>>> 'This float, {0:10.5f}, is fixed at 5 decimal places.'.format(3.1415926)
'This float, 3.14159, has width 0 and precision 5.'

>>> "Compare {0} and {0:0.20}".format(3.14)
'Compare 3.14 and 3.14000000000000001243'
```

String Formatting

- Numeric values are right-justified and strings are left- justified, by default.
- You can also specify a justification before the width.

```
>>> "left justification:
{0:<5}.format("Hi!") 'left justification:
Hi! '
>>> "right justification:
{0:>5}.format("Hi!") 'right justification:
Hi!'
>>> "centered: {0:^5}".format("Hi!")
'centered: Hi! '
```

Lists Methods

`l.append(item)` Add item at the end of a list

`l.insert(pos, item)` Add item at the specified position of a list

`l[pos] = value` Replace element at pos with value

`L[start:end] = sequence` Replace elements at pos start, up to but with elements in sequence

`l.remove(item)` Remove item in list

`l.pop(pos)` Remove item at pos in list

`l.clear()` Remove all items in list

`list(sequence)` Convert sequence to list

Files: Multi-line Strings

- A file is a sequence of data stored in secondary memory (disk drive).
- Files can contain any data type, but we focus on text.
- A file usually contains more than one line of text.
- Python uses the standard newline character (`\n`) to mark line

breaks.

Multi-Line Strings

```
Hello
World

Goodbye 32
```

- When stored in a file:
- `Hello\nWorld\n\nGoodbye 32\n`

`\n` affects print but not evaluation.

File Processing

- Opening a file associates the file on disk with an object in memory.
  - Once opened, the file is manipulated through this object.
- Closing the file completes any outstanding operations and bookkeeping for the file
  In some cases, not properly closing a file could result in data loss.

File Processing

- Associate a disk file with a file object using the open function
  - `<filevar> = open(<name>, <mode>)`
  - name is a string with the actual file name on the disk.
  - The mode is either ‘r’ or ‘w’ depending on whether we are reading or writing the file.

```
infile = open("numbers.dat",
"r") outfile =
open("mydata.out", "w")
```

File Methods

`<file>.read()` Returns the entire remaining contents of the file as a single (possibly large, multi-line) string

`<file>.readline()` Returns the next line of the file. This is all text up to and including the next newline character

`<file>.readlines()` Returns a list of the remaining lines in the file. Each list item is a single line including the newline characters.

`file.close()` Closes file and release resources

File Processing

```
infile = open(someFile, "r")
outfile = open("mydata.out", "w")

          for line in
   infile.readlines(): # Line
        processing here
    print(<expressions>, file=outfile)

infile.close()
outfile.close()
```

If an existing file is opened for writing, its contents will be cleared.

If the named file does not exist, a new one is created.

File Dialogs

- Python will look in the "current" directory for files if no path indicated.
- File names are in a form: <name>.<type> where type is a short indicator of what the file contains. E.g., `C:/users/susan/Documents/Python_Programs/users.txt`
- Alternatively, allow the users to browse the file system visually and navigate to the file

File Dialogs

- To ask the user for the name of a file to open, you can use `askopenfilename` from `tkinter.filedialog`.

```
from tkinter.filedialog import askopenfilena
…
infileName = aksopenfilename()
infile = open(infileName, "r")
```

File Dialogs

File Dialogs

- To ask the user for the name of a file to save, you can use `asksaveasfilename` from `tkinter.filedialog`.

```
from tkinter.filedialog import
asksaveasfilename
…
outfileName =
asksaveasfilename() outfile =
open(outfileName, "w")
```

File Dialogs

Simple Decisions

- Control structures allow us to alter sequential program flow.
- Decision structures allow program to execute different sequences of instructions for different cases, allowing the program to "choose" an appropriate course of action.

One-way Decisions

```
Input the temperature in
 degrees Celsius(call it
celsius) Calculate fahrenheit as
  9/5 celsius + 32
Output fahrenheit
If fahrenheit > 90
 print a heat warning
If fahrenheit > 30
 print a cold warning
```


One-way Decisions

```
if <condition>:
    <body>
```

- The body of the `if` either executes or not depending on the condition.
- In any case, control then passes to the next statement after the `if`.

One-way Decisions

```
def main():
 celsius = float(input("What is the Celsius
temperature? "))
 fahrenheit = 9 / 5 * celsius + 32
print("The temperature is", fahrenheit,
"degrees fahrenheit.")
 if fahrenheit >= 90:
 print("It's really hot out there, be
careful!")
 if fahrenheit <= 30:
 print("Brrrrr. Be sure to dress warmly")
```


Forming Simple Conditions

<expr> <relop> <expr>

<	<	Less than
<=	≤	Less than or equal to
==	=	Equal to
ndicates	≥	Greater than or equal to
>	>	Greater than
!=	≠	Not equal to

=

i

assign
ment

Forming Simple Conditions

Boolean conditions

- type `bool`
- values - true and false
  represented by the literals `True` and `False`.

```
>>> 3 < 4
True
```

```
>>> 3 * 4 < 3 + 4
False
>>> "hello" == "hello"
True
>>> "Hello" < "hello"
True
```


Logical Operators

The Boolean operators `and` and `or` are used to ~~combine two Boolean expressions and produce~~ a Boolean result.

```
<expr> and <expr>
<expr> or <expr>
```

T	T	T

T	F	F
F	T	F
F	F	F

T	F	T
F	T	T
F	F	F

T	T	T

Logical Operators

The `not` operator computes the opposite of a Boolean expression.

`not` is a *unary* operator, meaning it operates on a single expression.

T	F
F	T

Precedence of Logical Operators

Consider

```
a or not b and c
```

The order of precedence, from high to low, is `not`, `and`, `or`.

This statement is equivalent to

```
(a or ((not b) and c))
```


Two-Way Decisions

```
if <condition>:
 <statements>
else:
 <statements>
```


Two-Way Decisions

```
import math

def main():
 print "This program finds the real solutions to a
quadratic\n"
  a  =  float(input("Enter  coefficient  a:  "))
b = float(input("Enter coefficient b: "))  c
=  float(input("Enter  coefficient  c:  "))
discrim = b * b - 4 * a * c
 if discrim < 0:
 print("\nThe equation has no real roots!")
else:
 discRoot = math.sqrt(b * b - 4 * a * c)  root1 =
(-b + discRoot) / (2 * a)  root2 = (-b - discRoot) /
(2 * a)  print ("\nThe solutions are:", root1, root2
)
```

Multi-Way Decisions

```
if <condition1>:
 <case1
statements>
elif <condition2>:
 <case2
statements>
elif <condition3>:
 <case3
statements>
…
else:
 <default  statements>
```

Multi-Way Decisions

```
if discrim < 0:
 print("\nThe equation has no real roots!")
elif discrim == 0:
 root = -b / (2 * a)
 print("\nThere is a double root at", root)
else:
 discRoot = math.sqrt(b * b - 4 * a * c)  root1
= (-b + discRoot) / (2 * a)  root2 = (-b -
discRoot) / (2 * a)  print("\nThe solutions
are:", root1, root2 )
```

Study in Design: Max of Three

```
def main():
 print("Please enter three values,
separated by <ENTER>: ")
 x1 = int(input())
 x2 = int(input())
 x3 = int(input())

# missing code sets max to the value of
the largest

 print("The largest value is", maxval)
```

Strategy 1: Compare Each to All

This looks like a three-way decision, where we need to execute *one* of the following:

```
maxval = x1
maxval = x2
maxval = x3
```

All we need to do now is preface each one of these with the right condition such as:

```
if x1 >= x2 >=
x3:  maxval = x1
```

in most languages.
This condition is NOT right!

This syntax is not available

Strategy 1: Compare Each to All

We can separate these conditions with *and*!

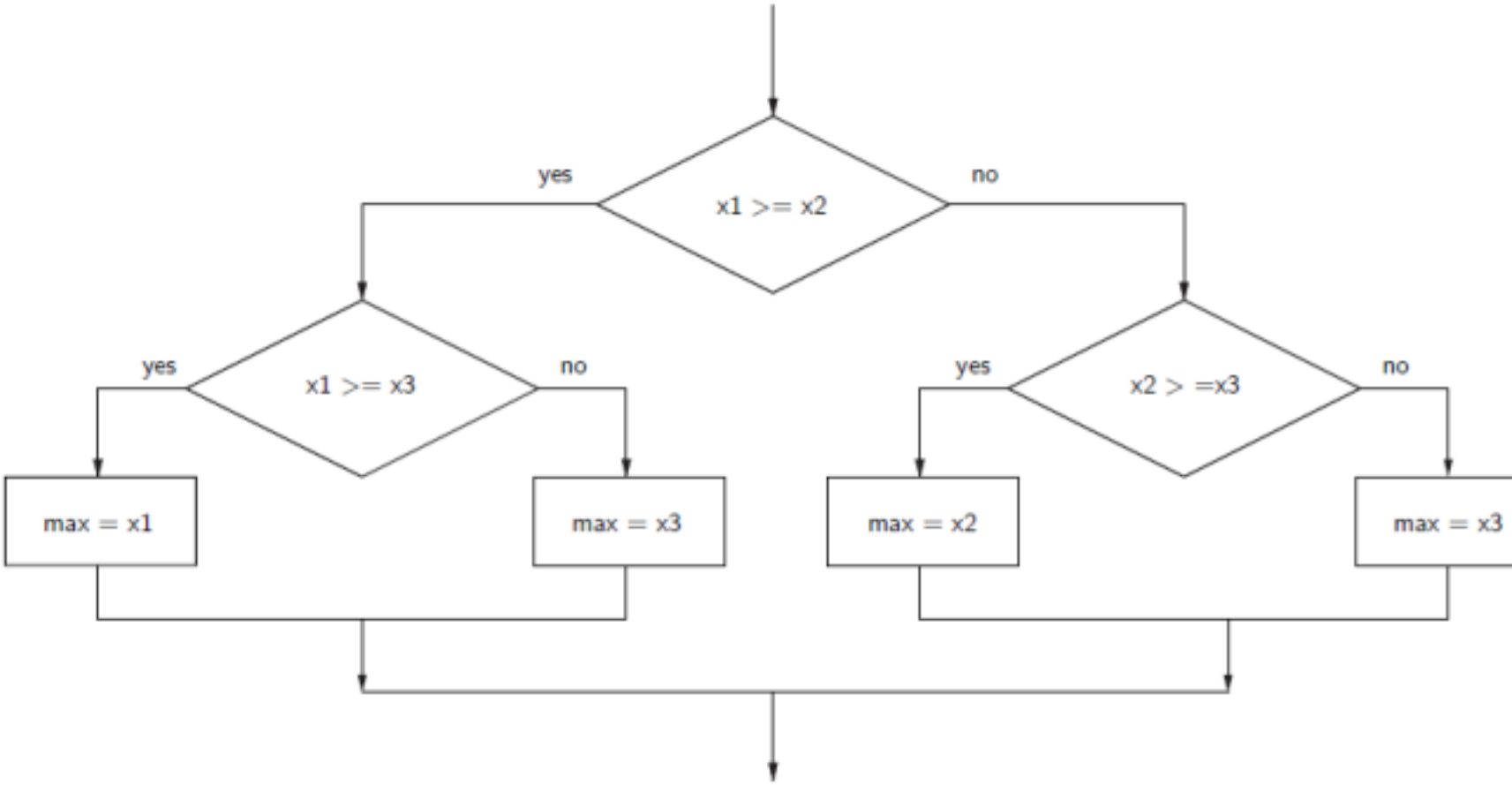
```
if x1 >= x2 and x1 >= x3:
 maxval = x1
elif x2 >= x1 and x2 >= x3:
 maxval = x2
else:
 maxval = x3
```

We're comparing each possible value against all thothers to determine which one is largest.

What would happen if we were trying to find

the max of five values?

Strategy 2: Decision Trees



Strategy 2: Decision Trees

```
if x1 >= x2:
 if x1 >= x3:
 maxval = x1
 else:
 maxval = x3
else:
 if x2 >= x3:
 maxval = x2
 else
 maxval = x3
```

Strategy 3: Sequential Processing

Strategy 3: Sequential Processing

```
maxval = x1
if x2 > maxval:
 maxval = x2
if x3 > maxval:
 maxval = x3
```

This process is repetitive and lends itself to using a loop. We prompt the user for a number, we compare it to our current max, if it is larger, we update the max value, repeat.

```
 for i in range(n-1):
 x = float(input("Enter a number >> "))
if x > max:
 max = x
```

Strategy 4: Library

Function

```
print("The largest value is",
      max(x1, x2, x3))
```