Organizing Data

• Sequences
  ➢ Tuples
  ➢ Lists

• Dictionaries

• Reading for this Lecture
  • Dawson, Chapter 4 (p. 104 to end) and 5
  • http://introcs.cs.princeton.edu/python/14array
Keeping Track of Data

• So far, in our programs, we have treated data in our programs as individual pieces, completely separate from one another
• This has worked for now, but as our programs become more complex, that will be impractical
• We are best served by finding ways to organize the data in our programs – so that we can keep track of it.
• We want to be able to create, use, and modify it in a predictable, logical manner.
Keeping Track of Data

• Python, like most programming languages, has a number of structures that will aid us in this.
• In some respects, as you will see, some of these structures are quite similar to one another.
• Despite this, they also have several differences, as well.
• For this reason, your program planning should include knowing what structure you are using…
• …and **why**!
Strings Revisited

• One very obvious (and common) form of data organization is the use of strings.

• After all, a string is actually a sequence of data points – specifically, characters.

• Organizing the characters into this form allows us to efficiently read, write, and modify text. We can:
  - Combine strings
  - Iterate (i.e., loop) through their characters
  - Extract single characters
  - Extract substrings
Sequences

• In fact, we can have sequences of any kind of data, regardless of type.

• In addition to the string form – a sequence of characters – we can also have sequences of:
  ➢ Numbers
  ➢ Booleans
  ➢ Strings
  ➢ Other sequences!

• After all, in Python, a sequence itself is an object

• In some programming languages, a sequence can contain only items of a particular type.

• Python, however, is more flexible in this, as we will see.
Tuples

• The most basic sequence in Python is probably the **tuple**
• A tuple is more or less just like a **string**, except that it can contain **any** kind of objects
• The syntax for creating a tuple is:

```python
variable = (first, second, ..., last)
```

• Examples:

```python
names = ("Bob", "Susan", "Jill")
id_numbers = (123, 456, 789)
booleans = (True, False, True)
items = ("Bob", 456, True)  # Items can be of different types, too
```
Tuples

• As with strings…
  • A tuple can be empty. `empty_tup = ()`
  • A tuple can be a condition. An empty tuple would be considered `False`, while a non-empty one would be considered `True`
  • You can print a tuple

```python
names = ("Bob", "Susan", "Jill")
print(names)
```

Prints as: `('Bob', 'Susan', 'Jill')`
Tuples

tuples = ("Bob", "Susan", "Jill")

• A tuple has a length. `len(names)` would evaluate to a result of 3

• You can loop through a tuple:

  ```python
  for name in names:
      print (name)
  ```

• You can concatenate tuples:

  ```python
  names += ("Bill", "Jack")
  print (names)
  ```

  Prints as: ('Bob', 'Susan', 'Jill', 'Bill', 'Jack')
Tuples

```python
names = ("Bob", "Susan", "Jill", "Bill", "Jack")

• You can use indices to get individual elements and slices of tuples, using the same syntax as with strings.

print("First item ": names[0])
print("Second item ": names[1])
print("Last item ": names[4])
print("Last item:", names[len(names)-1])

print (names[1:4])

Prints as: ('Susan', 'Jill', 'Bill')

• Just a slice of a string is a new string, a slice of a tuple is, in fact a new tuple

Bob
Susan
Jack
Jack
Tuples

• The other details about sequence positions – such as negative indices – also apply to tuples.

<table>
<thead>
<tr>
<th>names</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>'Bob'</td>
<td>'Susan'</td>
<td>'Jill'</td>
<td>'Bill'</td>
<td>'Jack'</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-5</td>
<td>-4</td>
<td>-3</td>
<td>-2</td>
<td>-1</td>
<td></td>
</tr>
</tbody>
</table>

• As with strings, a tuple is immutable. Even if the individual items within the tuple are mutable, the tuple itself – as a structure – is not.

• Tuple elements cannot be added, removed, or replaced.

• As with strings, the most you can do is create a new tuple out of other, existing ones.
Examples using tuples:

- **hero’s_inventory.py**
  - Simple example of the creation and use of tuples – namely, printing the tuple as a whole versus its individual elements

- **hero’s_inventory2.py**
  - More complex example illustrating:
    - Use of `len()` function
    - Use of `in` operator
    - Indexing
    - Slicing
    - Concatenation

- **word_jumble.py**
  - Extended example of developing a program for a word game
Lists

• One major limitation of tuples is their immutability

• It would be nice to have a sequence that you can actually change, rather than simply creating a new one each time

• Python also has a mutable sequence, in the form of the list – a structure very similar to tuples, but with many important differences.
Creating Lists

• If you recall, you would create a *tuple* this way:

```python
names = ("Bob", "Susan", "Jill")
```

• In contrast, you would create a *list* this way:

```python
names = ["Bob", "Susan", "Jill"]
```

• In other words, the only difference in the syntax for creating is the pair of symbols *encasing the sequence*:

```
variable = (first, second,..., last)
```

```
variable = [first, second,..., last]
```
```
names = ["Bob", "Susan", "Jill"]
```

**len() function:**

```
print (len(names))
```

3

**in operator:**

```
print ("Bob" in names)
```

True

**concatenate:**

```
names += ["Bill", "Jack"]
```

(new list)

**get an item:**

```
print (names[2])
```

Jill

**get a slice:**

```
print (names[1:4])
```

["Susan","Jill","Bill"]
List Mutability

names = [“Bob”, “Susan”, “Jill”, “Bill”, “Jack”]

• However, the fact that lists are **mutable** means they have some additional options

Replace an item:

```python
```

Replace a slice:

```python
```

Delete an item:

```python
del names[2]
```

Delete a slice:

```python
del names[1:4]
```
List Use and Methods

• We can see *list mutability* in action in the program `hero’s_inventory3.py`.

• In addition, Python has several functions/methods you can use for manipulating lists.

• See `high_scores.py`.

• The list methods used in that program – along with other methods – are in the textbook in Table 5.1 on page 132.
Dictionaries

• In addition to sequences, another useful way to organize data is in terms of *key-value pairings*.

• This is the case with a **dictionary**, where data is organized like so:

\[
\begin{align*}
  \text{key1} & \rightarrow \text{value1} \\
  \text{key2} & \rightarrow \text{value2} \\
  \text{key3} & \rightarrow \text{value3} \\
  \vdots
\end{align*}
\]

• You can then use a specific **key** to retrieve a particular **value** from the dictionary.
Creating Dictionaries

- **Syntax:**

  ```
  variable = { first_key : first_value, 
               second_key : second_value, 
               ... 
               last_key     : last_value }
  ```

- Keys must be of an **immutable** type, but values can be of **any** type

- Each key in the dictionary **must be unique**; otherwise, duplicated keys would create ambiguity
Using Dictionaries

• Let’s create a dictionary:

```python
info = { "name" : "John Doe",
        "school" : "UMB",
        "ID" : 12345,
        "GPA" : 3.7 }
```

• Now, we can…

Fetch a value by key:

```python
print("My name is: " + info["name"])
```

```python
My name is: John Doe
```

See if key exists:

```python
print("Has major: " + str("major" in info))
```

```python
Has major: False
```
### Using Dictionaries

#### info

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>John Doe</td>
</tr>
<tr>
<td>school</td>
<td>UMB</td>
</tr>
<tr>
<td>ID</td>
<td>12345</td>
</tr>
<tr>
<td>GPA</td>
<td>3.7</td>
</tr>
</tbody>
</table>

**Add a new entry** *(key-value pair)*:

```python
info["major"] = "Comp. Sci."
```

**Replace an entry**:

```python
info["major"] = "Art"
```
Using Dictionaries

Delete an entry by key:

```python
del info["major"]
```

Fetch a value by key (with default):

```python
print("Major:" info.get("major", "Undeclared"))
```
Dictionary Use and Methods

• We can see an extended example in the program `geek_translator.py`

• This program depicts the use of a dictionary to organize data about words and their definitions

• We see the **dynamism** of the structure

• Other dictionary methods can be seen in the textbook in **Table 5.2 on page 148.**
Nested Structures

• We stated earlier that tuples, lists, and dictionaries can hold values of any type.

• This means that those values can actually be other tuples, lists, and dictionaries!

• Nested structures can be very useful for keeping track of many pieces of data that are related to one another in some respect.

• Consider `high_scores2.py`
Example: Nested Dictionaries

book =
{
    "title"   : "How to Program",
    "author"  : "John Doe",
    "pub_year" : 2016,
    "chapters" : {
        1 : "Printing Text",
        2 : "Making Strings",
        3 : "Using Variables"
    },
    "price"  : 27.50
}

• Variable **book** refers to a dictionary with the keys "title", "author", "pub_year", "chapters", and "price"

• However, the value at **book["chapters"]** is another dictionary, with the keys 1, 2, and 3
Example: Nested Dictionaries

```python
book = {
    "title" : "How to Program",
    "author" : "John Doe",
    "pub_year" : 2016,
    "chapters" : {
        1 : "Printing Text",
        2 : "Making Strings",
        3 : "Using Variables"
    },
    "price" : 27.50
}
```

- To get the title of the third chapter, we would use the following expression:
  ```python
  book["chapters"] [3]
  ```
- We could also add a fourth chapter:
  ```python
  book["chapters"][4] = "Writing Expressions"
  ```
Example: A Tuple of Dictionaries

books = (  
    {  
        "title": "How to Program",  
        "author": "John Doe",  
        "pub_year": 2016,  
        "price": 27.50  
    },  
    {  
        "title": "Calculus",  
        "author": "Jane Doe",  
        "pub_year": 2015,  
        "price": 39.95  
    },  
    {  
        "title": "Biology",  
        "author": "Jim Doe",  
        "pub_year": 2016,  
        "price": 87.29  
    }  
)  

print (books[1]["price"])  
print (books[2]["pub_year"])