Program Planning, Data Comparisons, Strings

- Program Planning
- Data Comparisons
- Strings
- *Reading for this class:*

  Dawson, Chapter 3 (p. 80 to end) and 4

Program Planning

- When you write your first programs, there may not be much planning involved because they are so simple. You just sit down and start typing!
- However, as you start to tackle more complex tasks, it will become ever more import to think about the problem and do some planning first.
- Doing so can make your job easier and save you lots of time and effort, later on.

Program Planning

- Remember, a program is ultimately a *series of steps* (also known as an “algorithm”) for *accomplishing a task*
- One important program-planning skill to develop is how to write and use *pseudocode*, which is essentially the steps of an algorithm written in human language instead of code.
- This is helpful on two levels:
    1. It gets you to thinking about the direction you want your program to take and encourages you to think things through before getting started.
    2. You will get into the habit of thinking about computations in a more abstract and conceptual manner – instead of thinking in a specific programming language

Pseudocode Example: Computing a Sum

As pseudocode:

```
Start with a sum of zero
While the sum is less than or equal to 100
        Get a new integer value from the user
        Add the new value to the sum
        Print the current sum
Print the final sum
```

As Python code:

```
sum = 0
while sum <= 100:
     new_value = int(input("Type an integer: "))
     sum += new_value
     print ("The sum is currently:", sum)
print (“\nThe sum is:", sum)
```

Program Planning

- When we, as humans, carry out a task, our minds tend to leave many aspects of the process implicit. We take them for granted and do not think of them.
- In fact, these tasks tend to involve numerous smaller steps – sometimes tiny ones – that do not immediately occur to us
- We do not have to think of them explicitly
- In programming, however, you **must** be explicit
- When you start writing a program, you are likely thinking in terms of the program’s behavior when running
- As such, your steps may be rather wide and general

Program Planning

- As such, you must take your initial wide and general steps and *break them down* into the smaller steps that make them up
- This is called **stepwise refinement**. The basic process is to look at a step and see if it easily can be translated into a single line of code.
- If not, you can refine the step some more to get a set of smaller steps. It is largely about learning to “think like a computer”.
- See the textbook example, pages 81 to 84. It shows multiple steps of the program planning process

Comparing Data

- When comparing data using boolean expressions, it's important to understand the peculiarities of certain data types

- Let's examine some key situations:
  - Comparing double/float values for equality
  - Comparing characters
  - Comparing strings (alphabetical order)

Comparing Decimals

- The equality operator (==) is not always the best choice for comparing two decimals (`float` type)

- They are equal **<u>only</u>** if their underlying binary representations match exactly

- However, in real life, it is rarely necessary for two figures to be absolutely equal

- Two decimals may be "close enough," even if they aren't exactly equal, yet computations often result in slight differences that may be irrelevant

How To Compare Decimals

- Decide on a "maximum tolerable inequality":

```
TOLERANCE = 0.000001
```

- To determine the equality of two decimals, use the following technique:

```
if abs(d1 - d2) < TOLERANCE:
   print ("Essentially equal")
```

- If **the absolute value of the difference** is less than the tolerance, the *if-condition* will be true, and the print statement will execute. (The idea here is "equal enough")

- The size of the tolerance will differ, depending on the problem at hand.

Comparing Characters

- As we've discussed, Python uses the Unicode character set
- Each character has a particular numeric value, which creates an ordering of characters
- Thus, we can use relational operators on character data
- For example, `'A' < 'J' == True` because 'A' has the smaller numeric value in the Unicode set

Comparing Characters

- In Unicode, the digit characters (0-9) are contiguous and in order of their numerical value

- Likewise, the uppercase letters (A-Z) and lowercase letters (a-z) are contiguous and in alphabetical order

Characters	Unicode Values
0 – 9	48 through 57
A – Z	65 through 90
a – z	97 through 122

- Notice that *uppercase precedes lowercase*!

Comparing Characters

- Therefore, we can determine whether a character is a digit, a letter, etc.

```
if character >= '0' and character <= '9':
    print ("Yes, it's a digit!")
elif ((character >= 'A' and character <= 'Z') or \
         (character >= 'a' and character <= 'z')):
    print ("It's a letter!")
else:
    print ("Something else entirely!")
```

Comparing Strings

- We can also use the == operator to determine if the values of two strings are identical (character by character):

```
if name1 == name2:
   print ("Same name")
```

- This also applies to the other equality and relational operators:
  - !=
  - <
  - <=
  - >
  - >=

```
name1 = "Bill"
name2 = "Bob‶

name1 == name2
                False
name1 <= name2
                 True
name2 > name1
                 True
```

Comparing Strings

```
if name1 < name2:
   print (name1 + "comes first")
else:
   if name1 == name2:
      print ("Same name")
   else:
      print (name2 + "comes first")
```

- Results may sometimes surprise you!

- The comparison is based on characters' **numeric** values, so it is called a *lexicographic ordering*

Lexicographic Ordering

- Lexicographic ordering is not strictly alphabetical
- For example, the string `"Great"` comes before the string `"fantastic"`. In Unicode, the uppercase letters have lower values than lowercase, so 'G' is technically less than 'f'
- Also, short strings come before longer strings with the same prefix
- `"book"` comes before `"bookcase"`, but `"Bookcase"` comes before **both**!

Using Strings

- Because strings will be a huge part of your programming experience, it's important to become more familiar and comfortable with their workings.
- Moreover, this is important preparation for other kinds of sequences.
- In particular, the techniques described and demonstrated here are ones that you should practice and remember

See message_analyzer.py

Using Strings

- The **<u>len()</u>** function – gives you the length of a sequence, such as a string:

```
message = “hello”
print (“The length of the string is”, len (message))
```

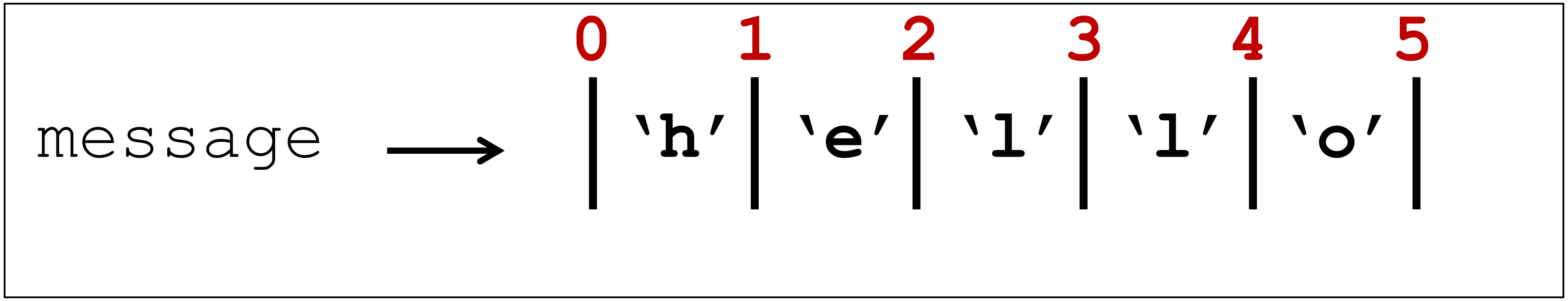
- The **<u>in</u>** operator. In a <u>for loop</u>, this is used to provide the items in a sequence. However, it can *<u>also</u>* tell you if a sequence does (**`True`**) or does not (**`False`**) contain a particular item:

```
print(“The string contains an ‘e’:”, ‘e’ in message)
print(“The string contains an ‘A’:”, ‘A’ in message)
OUTPUT:
The string contains an ‘e’: True
The string contains an ‘A’: False
```

Indexing Strings

- Because a string is a sequence, characters can be accessed by position numbers
- A string’s characters are numbered from **<u>zero</u>** to the <u>length minus one</u>. Think of it like this:

```
message = “hello”
```



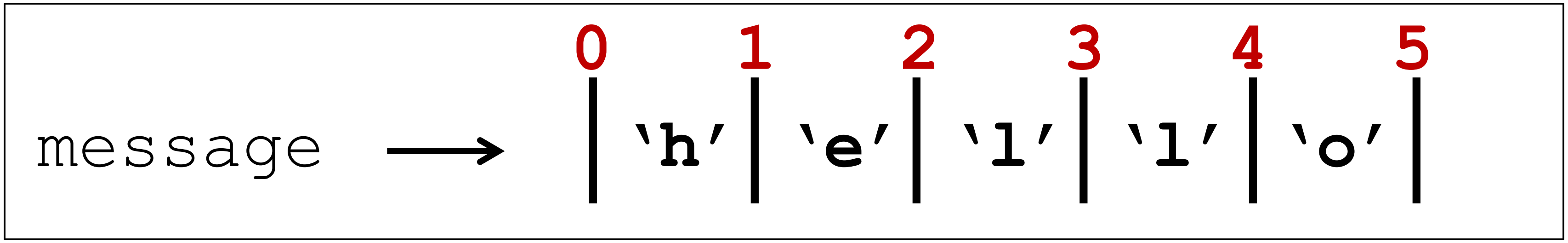
0 1 2 3 4 5

`message` → | ‘h’ | ‘e’ | ‘l’ | ‘l’ | ‘o’ |

See random_access.py

Indexing Strings

```
message = “hello”
```



- To get a the character at a position within a string, you use the following syntax:

```
the_string[position]
```

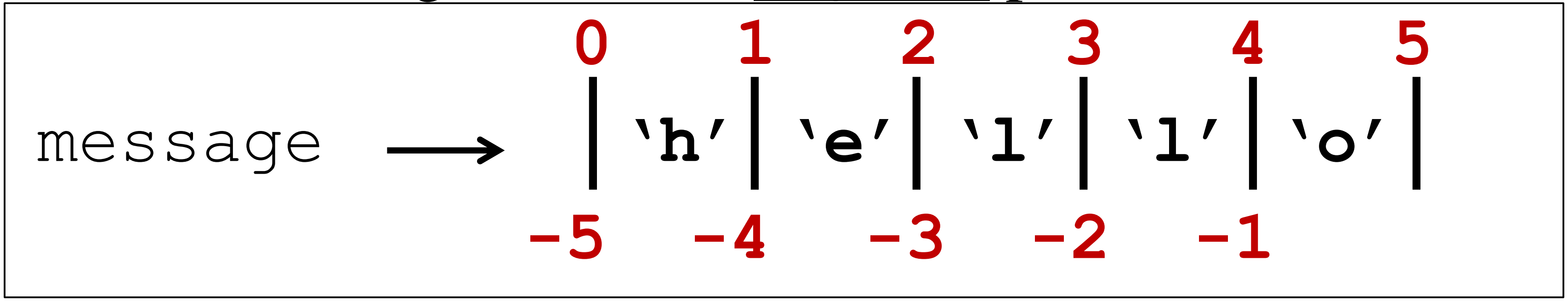
```
print(“First character:”, message[0])
print(“Second character:”, message[1])
print(“Last character:”, message[4])
print(“Last character:”, message[len(message)-1])
```

First character position: 0

If you don't already know the string length

Indexing Strings

- In fact, strings also have **negative** position numbers:



- Thus, the following code would **also** work:

```
print("First character:", message[-5])
print("Second character:", message[-4])
print("Last character:", message[-1])
```

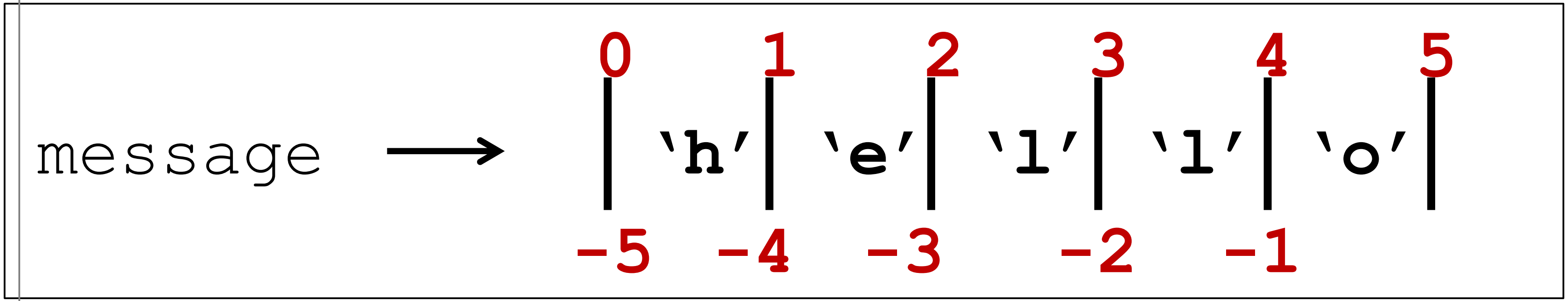
Last character position: -1

OR

```
print("First character:", message[0 - len(message)])
print("Second character:", message[1 - len(message)])
```

Slicing Strings

- In addition, you can use indices to get a *subsection* of a string, called a **slice**



- To get a slice, you use the following syntax:

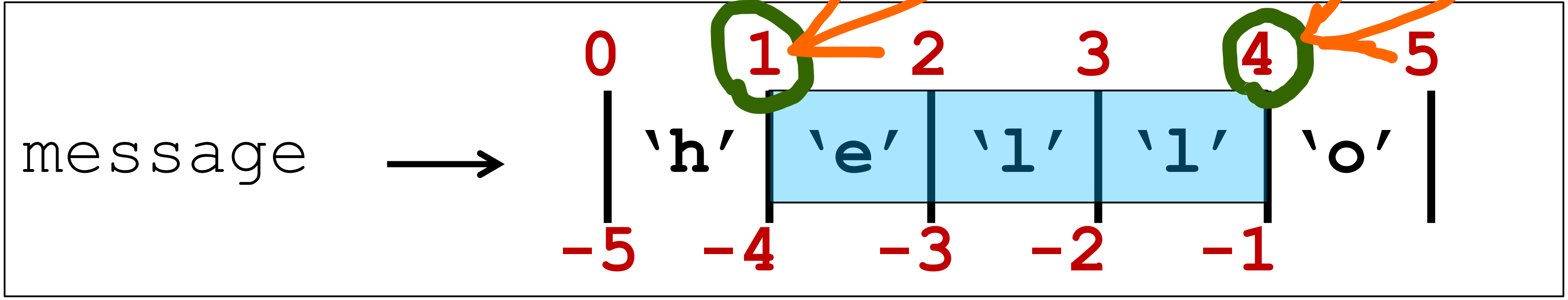
```
the_string[start:end]
print(“First two:”, message[0:2])
print(“Middle three:”, message[-4:-1])
print(“Last two:”, message[-2:5 ])
print(“Last character:”, message[len(message)-1])
```

Why does this work?

See pizza_slicer.py

Slicing Strings

- You may notice something about slice syntax. Specifically, we seem to ***start*** with the position of the *first* character of the slice but ***end*** with the position *one greater than* the last character



- *start* and *end* indicate the slice ***boundaries***:

```
the_string[start:end]
```

```
print(“Middle three:”, message[1:4])
```

We could have also used -4:-1

will print as:

```
ell
```

String Immutability

- The term “*mutable*” indicates that something can be changed or altered – versus “***im**mutable*”, which cannot be changed
- Strings are one example of this. A string, once created, is unchangeable.
- A line of code like `message += “ world!”` might *appear* to change `“hello”` into `“hello world!”`
- Actually, a **new** string is created from the two old ones and then reassigned to `message`
- Similarly, `message[1:4]` is actually a **new** string created from `message`