Monday, April 16, 2018

Unit 3: Python (cont'd)

Readings in *PCfB:*

Ch 7 ("Components of Programming")

Ch 9 ("Decisions and Loops")

Topics:

- more operations on strings, string methods (concepts you'll need for Project 3)
- in-class projects: `gc_content.py`, `gp.py`
- executable files
- `if` statements
- Boolean operators
- notes on programming style

Review: Strings

Previously we learned how to create strings in Python:

```
In [1]: s1 = 'aloha'

In [2]: s2 = 'Area 51'
```

And we learned about functions and operators that work with strings:

```
In [3]: len(s1)
Out[3]: 5

In [4]: len(s2)
Out[4]: 7

In [5]: s1 + "!"
Out[5]: 'aloha!'
```

Square brackets (also known as the “index operator”) are used to access characters in a string:

```
In [6]: s2[0]
Out[6]: 'A'
```

Important: *The first character is at* `s[0]` *(not* `s[1]`*)*

Methods

Most functions that work with strings use an alternative syntax

Instead of writing

```
f(s)
```

we write

```
s.f()
```

Simply take the string name out of the parentheses and put it before the function name, with a period between the string and function name

Functions that are called using this syntax are referred to as **methods**

Example

```
In [12]: s = 'nice!'

In [13]: s.upper()
Out[13]: 'NICE!'
```

Strings are Immutable

The call to upper did not change s — it returned a **copy** of s, with all letters set to upper case

```
In [14]: s
Out[14]: 'nice!'

In [15]: t = s.upper()

In [16]: print(s,t)
nice! NICE!
```

Often a program will “update” a string by throwing away the old version and replacing it with a new one:

```
In [17]: s = s.upper()

In [18]: s += '!'

In [19]: s
Out[19]: 'NICE!!'
```

More Methods

Here is a method used in an example program in *PCfB*

- call `s.count(t)` to find the number of times `t` occurs in `s`

```
In [20]: seq = 'GATTACA'

In [21]: seq.count('A')
Out[21]: 3

In [22]: seq.count('C')
Out[22]: 1
```

See dnacalc.py *in PCfB*

We can pass any substring to count, and it will return the number of times that substring is found

```
In [25]: lyric = 'fa la la la la, la la la la'

In [26]: lyric.count('la')
Out[26]: 8

In [27]: lyric.count('foo')
Out[27]: 0
```

`startswith` and `endswith`

These two methods check to see if s string starts or ends with a specific substring

```
In [32]: lyric
Out[32]: 'fa la la la la, la la la la'

In [33]: lyric.startswith('fa')
Out[33]: True

In [34]: lyric.startswith('la')
Out[34]: False

In [35]: lyric.endswith('la la')
Out[35]: True

In [36]: lyric.endswith('la la land')
Out[36]: False
```

> The values returned by `startswith` and `endswith` are **Boolean** values
>
> They're named for George Boole (1815--1864), a pioneer in the field of symbolic logic
>
> If the result of evaluating an expression is `True` or `False` we say it is a **Boolean expression**

In-Class Project

Let's do an in-class programming project

Write a program named `gc_content.py` that will compute the GC content of a strand of DNA

[see Exercise.pdf]

Slices

The index operator also allows us to select a **range** of characters

The notation **`s[i:j]`** means "all the characters from position `i` up through position `j-1` in s"

```
In [1]: s = 'abcdefghij'

In [2]: len(s)
Out[2]: 10

In [3]: s[0]
Out[3]: 'a'
```

Individual letters are `s[0]` *through* `s[9]`

```
In [4]: s[0:4]
Out[4]: 'abcd'
```

The first 4 letters in s

```
In [5]: s[3:8]
Out[5]: 'defgh'
```

Letters 4 through 8

```
In [6]: s[3:]
Out[6]: 'defghij'
```

All the letters from position 3 on

```
In [7]: s[:5]
Out[7]: 'abcde'
```

From the beginning up to letter 5

Hint: you'll want to use slices in the "Pig Latin" program on Project 3

Conditional Execution

Suppose we want to write a program that creates the plural form of a word

It's simple to write a statement that appends `'s'` to the input:

```
word = argv[1]
word += 's'
print(word)
```

Here's the output from some test cases:

```
$ python plural.py duck
ducks
```

```
$ python plural.py cat
cats
```

But as we know English isn't that regular:

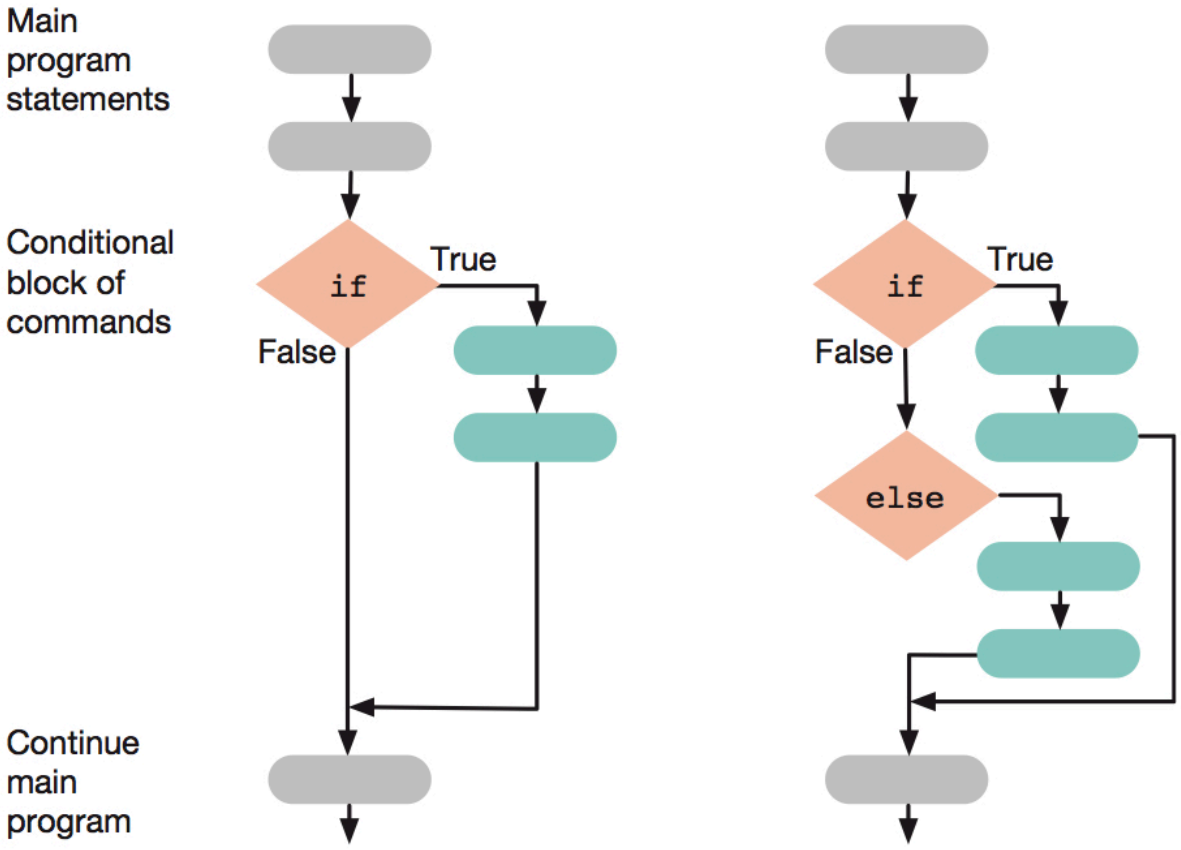
```
$ python plural.py fish
fishs
```

To solve this problem we need a form of **conditional execution**

- tell Python to test some condition ("does the word end with an h?")
- if so append "es", otherwise append "s"

Flow Chart

Figure 7.3 from *PCfB* shows the general idea:



`if` Statements

In Python conditional execution is controlled by an **`if` statement**

Here's how we can append "es" if a word ends with "h":

```
if word.endswith('h'):
    word += 'es'
else:
    word += 's'
```

See `plural.v1.py`

The Details

An `if` statement starts with the word `if`

- immediately following `if` there should be a Boolean expression (something that evaluates to True or False)
- note the **colon** at the end of the expression
- there can be any number of statements in the **body** of the `if` statement

Important: the statements in the body **must be indented**, and they must all be indented *exactly* the same [the "standard" indent is 4 spaces]

The **`else` statement** is optional (see the flow chart on the previous page)

- note the colon following `else` and the statements in the body are indented

Another Rule for `plural.py`

Here's an example of where our current program won't produce the right answer:

```
$ python plural.py company
companys
```

We can fix this problem by adding another rule: if a word ends with "y" replace the "y" with "ies"

Before we start coding we should do some **interactive experiments** to see how to find the end of a word and how to change it

```
In [1]: s = 'company'
```

The index operator (square brackets) should tell us which letter is at the end:

```
In [2]: s[len(s)]
IndexError: string index out of range
```

Oops. Indexes range from 0 to 1 less than the length (in this case 0 to 6):

```
In [3]: s[len(s)-1]
Out[3]: 'y'
```

As a shortcut we can use negative numbers to index from the right:

```
In [3]: s[-1]
Out[3]: 'y'
```

Another Rule (cont'd)

Let's see if we can change what's there now to "ies"

```
In [4]: s[len(s)] = 'ies'
TypeError: 'str' object does not support item
assignment
```

Bummer! Python string are immutable — we can't change them.

> *Don't be misled by* `word += 's'`
> *Python makes a new string and replaces the old value of* `word`

A New Strategy

We can make "companies" by using the + operator: make a substring using all but the last character in s and append "ies"

The slice operator will give us the substring:

```
In [5]: s[0:len(s)-1]
Out[5]: 'compan'
```

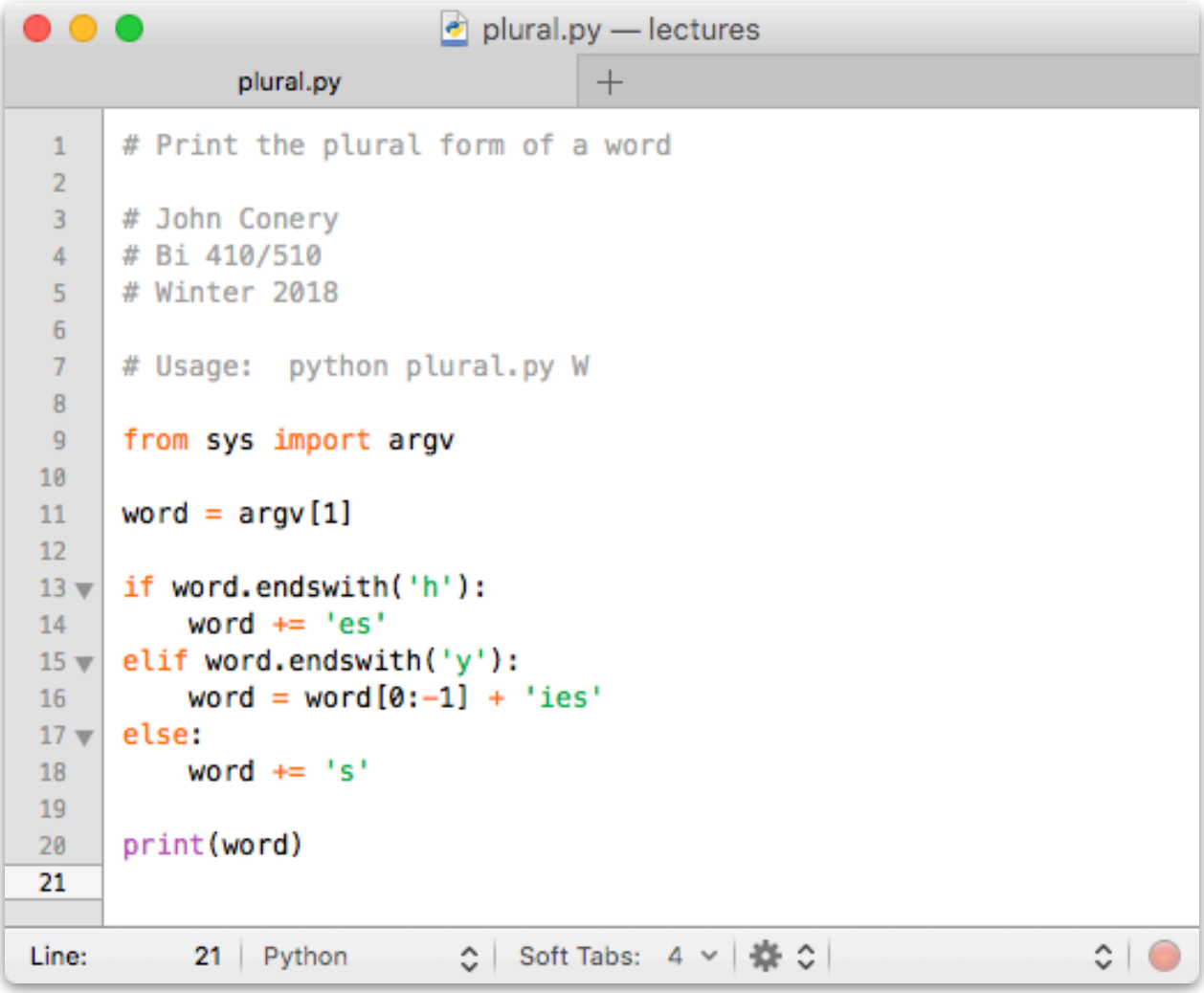
We can use negative indexes in slices, too:

```
In [6]: s[0:-1]
Out[6]: 'compan'
```

`elif`

Here is the new program.

It also introduces a new statement: `elif`. The name is a combination of "else" and "if":



Style Notes

Note again that this program gets the values it needs from `argv` at the beginning of the program

- there is no reference to `argv` after line 11

Note also the general outline:

- get input values from `argv`
- do the calculations
- print the results

We could have print statements inside the bodies of the `if`/`elif`/`else` statements, but this design lets us change the output format more easily

- try to keep the code that does I/O separate from the main program logic

Beware `elif`

A program that has a long "cascade" of `if`/`elif` statements can often be rewritten in a better style — we'll show an example below and throughout the term

Running the New Version

Here are some tests of the latest version of the program

The first two tests are **regression tests**: make sure previous cases still work and we didn't break anything:

```
$ python plural.v2.py duck
ducks
```

```
$ python plural.v2.py cat
cats
```

Test the new rule:

```
$ python plural.v2.py fish
fishes
```

```
$ python plural.v2.py finch
finches
```

But we still have work to do:

```
$ python plural.v2.py moth
mothes
```

```
$ python plural.v2.py albatross
albatrosss
```

Testing Multiple Conditions

A more accurate rule for adding “es” to a word is to see if the word ends with “s”, “x”, “ch”, or “sh”

We could write this using `elif`:

```
if word.endswith('s'):
    word += 'es'
elif word.endswith('x'):
    word += 'es'
elif word.endswith('ch'):
    word += 'es'
elif word.endswith('sh'):
    word += 'es'
```

Bad form — the bodies are all identical.

Can we combine them into a single case?

Boolean Operators

We can combine tests using Boolean operators

`x and y`	True if both x and y are True
`x or y`	True if either x or y (or both) are True
`not x`	True is x is False

Using the `or` operator in `plural.py`:

```
if word.endswith('x') or word.endswith('s')
      or word.endswith('ch') …
```

[we'll leave it as an exercise to revise and test the program with this Boolean expression]

Comparison Operators

Python has a number of ways to compare number and strings (and other types of data)

Here are some examples —

```
In [1]: n = 10

In [2]: m = 20

In [3]: n > 0
Out[3]: True

In [4]: m < 100
Out[4]: True

In [5]: n > 0 and m < 100
Out[5]: True

In [6]: n > 0 and m < 10
Out[6]: False

In [7]: n > 0 or m < 10
Out[7]: True

In [8]: n == 10
Out[8]: True

In [9]: n == 0
Out[9]: False
```

NOTE! *To see if two items are the same use a double equal sign*

(the single equal sign is the assignment operator)

List of Boolean Operators

Table 7.3 from *PCfB*:

Comparative	
Equal to	==
Not equal to	!=, <>, ~=
Greater than	>
Less than	<
Greater or equal	>=
Less or equal	<=
Logical	
And	and &, &&
Or	or, |, ||
Not	not !, ~

Don't use these — they're special purpose operators defined for integers

In-Class Project

Let's write a program that converts letter grades to point values:

```
$ python gp.py B
3

$ python gp.py A
4
```

Next week we'll extend the program so it computes a GPA based on several grades:

```
$ python gp.py B A
3.5
```