

CS2204

Program Design and Data
Structures for Scientific
Computing

Announcements

- Our first programming assignment is ready; but we need to cover more material before we are ready to tackle it

True and False

- **True** and **False** are constants in Python.
- Other values equivalent to **True** and **False**:
 - **False**: zero, **None**, empty container or object
 - **True**: non-zero numbers, non-empty objects
- Comparison operators: `==`, `!=`, `<`, `<=`, etc.
 - Operators can be chained
`X <= Y <= Z`
 - X and Y have same value: `X == Y`
 - Compare with `X is Y`
 - X and Y are two variables that refer to the *identical same object*.

Boolean Logic Expressions

- You can also combine Boolean expressions.
 - True if a is true and b is true: `a and b`
 - True if a is true or b is true: `a or b`
 - True if a is false: `not a`
- Use parentheses as needed to disambiguate complex Boolean expressions.

For Loops

- A for-loop steps through, or iterate over, each of the items in a sequence (such as a list, tuple, or string), or any other type of object which is “*iterable*”.

```
for <item> in <collection>:  
    <statements>
```

- If `<collection>` is a list or a tuple, then the loop steps through each element of the sequence.
- If `<collection>` is a string, then the loop steps through each character of the string.

```
for someChar in "Hello World":  
    print(someChar)
```

For Loops - continued

- Since a variable often ranges over some sequence of numbers, the `range()` function returns a list of numbers from 0 up to but not including the number we pass to it.
- `range(5)` returns the list `[0,1,2,3,4]`
- So we could say:

```
for x in range(50):  
    print(x, ": I love CS!")
```

- (There are more complex forms of `range()` that provide richer functionality...)

In-Class Exercise

Prompt the user for 10 inputs

Print out the following for the dataset:

- **Min**
- **Max**
- **Sum**
- **Average**

Rules:

- **Least lines of code wins**
- **Group size ≤ 2**
- **If/While/For, int, float only (no array, list, tuple, etc.)**
- **PEP8 Compliant!!!**

The String Data Type

- The most common use of personal computers is word processing.
- Text is represented in programs by the *string* data type.
- A string is a sequence of characters enclosed within quotation marks (") or apostrophes (').

```
>>> str1="Hello"  
>>> str2='spam'  
>>> print(str1, str2)  
Hello spam  
>>> type(str1)  
<type 'str'>
```


The String Data Type

- Getting a string as input

```
>>> firstName = raw_input("Please enter your name: ")
Please enter your name: John
>>> print("Hello", firstName)
Hello John
```

- **raw_input()** will print the prompt (no linefeed) then will read a line of data and return it as a string

The String Data Type

- We can access the individual characters in a string through *indexing*.
- The positions in a string are numbered from the left, starting with 0.
 - Zero-based indexing just like Java and C++
- The general form is `<string>[<expr>]`, where the value of `expr` determines which character is selected from the string.

The String Data Type

H	e	l	l	o		B	o	b
0	1	2	3	4	5	6	7	8

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

The String Data Type

H	e	l	l	o		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.
- We can index from the right side using negative indexes.

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

The String Data Type

H	e	l	l	o		B	o	b
0	1	2	3	4	5	6	7	8

- Python strings are **immutable**
(can't change individual chars)

```
>>> greet[0] = 'J'    # this will fail
```

- All string operations (concatenation, etc.) return new strings that are also immutable

The String Data Type

- Indexing returns a string containing a single character from a larger string.
- We can also access a contiguous sequence of characters, called a *substring*, through a process called *slicing*.
- Slicing:
`<string>[<start>:<end>]`
- start and end should both be ints
- The slice contains the substring beginning at position start and runs up to **but doesn't include** the position end.
- If either expression is missing, then the start or the end of the string are used.

The String Data Type

H	e	l	l	o		B	o	b
0	1	2	3	4	5	6	7	8

```
>>> greet[0:3]
```

```
'Hel'
```

```
>>> greet[5:9]
```

```
' Bob'
```

```
>>> greet[:5]
```

```
'Hello'
```

```
>>> greet[5:]
```

```
' Bob'
```

```
>>> greet[:]      # make a copy of entire string
```

```
'Hello Bob'
```

The String Data Type

- Can we put two strings together into a longer string?
- *Concatenation* “glues” two strings together (+)
- *Repetition* builds up a string by multiple concatenations of a string with itself (*)

The String Data Type

- The function *len* will return the length of a string.
- And we saw earlier how the *for* loop can iterate over the characters of a string

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

```
4
```

```
>>> for ch in "Spam!":  
    print ch
```

```
S
```

```
p
```

```
a
```

```
m
```

```
!
```

The String Data Type

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

Simple String Processing

- Usernames on a computer system
 - First six characters of last name, then first initial, and then middle initial

```
# get user's first, middle, and last names
first = raw_input("Please enter your first name (lowercase): ")
middle = raw_input("Please enter your middle name (lowercase): ")
last = raw_input("Please enter your last name (lowercase): ")

# concatenate 6 chars of last name with first and middle initials
vunetid = last[:6] + first[0] + middle[0]
```

Simple String Processing

- Another use – converting an int that stands for the month into the three letter abbreviation for that month.
- Store all the names in one big string:
`"JanFebMarAprMayJunJulAugSepOctNovDec"`
- Use the month number as an index for slicing this string:
`monthAbbrev = months[pos:pos+3]`

Simple String Processing

Month	Number	Position
Jan	1	0
Feb	2	3
Mar	3	6
Apr	4	9

- To get the correct position, subtract one from the month number and multiply by three

Simple String Processing

```
# months is used as a lookup table
months = "JanFebMarAprMayJunJulAugSepOctNovDec"

n = int(raw_input("Enter a month number (1-12): "))

# compute starting position of month n in months
pos = (n-1) * 3

# Grab the appropriate slice from months
monthAbbrev = months[pos:pos+3]

# print the result
print("The month abbreviation is", monthAbbrev + ".")
```

Simple String Processing

- One weakness – this method only works where the potential outputs all have the same length.
 - Each month abbreviation is three characters long
- How could you handle spelling out the months?
 - We will be able to solve this problem when we see lists
- It turns out that strings are really a special kind of *sequence*, so these operations also apply to sequences!
 - Lists are also a kind of sequence, as we will see

Strings Methods

- One of these methods is *split*. This will split a string into substrings based on spaces.

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

Strings Methods

- Split can be used on characters other than space, by supplying the character as a parameter.

```
>>> "32,24,25,57".split(",")  
['32', '24', '25', '57']  
>>>
```

Other String Methods

- There are a number of other string methods. Try them all!
 - `s.capitalize()` – Copy of `s` with only the first character capitalized
 - `s.title()` – Copy of `s`; first character of each word capitalized
 - `s.center(width)` – Center `s` in a field of given width

Other String Operations

- `s.count(sub)` – Count the number of occurrences of `sub` in `s`
- `s.find(sub)` – Find the first position where `sub` occurs in `s`
- `s.join(list)` – Concatenate list of strings into one large string using `s` as separator.
- `s.ljust(width)` – Like center, but `s` is left-justified

Other String Operations

- `s.lower()` – Copy of `s` in all lowercase letters
- `s.lstrip()` – Copy of `s` with leading whitespace removed
- `s.replace(oldsub, newsub)` – Replace occurrences of `oldsub` in `s` with `newsub`
- `s.rfind(sub)` – Like `find`, but returns the right-most position
- `s.rjust(width)` – Like `center`, but `s` is right-justified

Other String Operations

- `s.rstrip()` – Copy of `s` with trailing whitespace removed
- `s.split()` – Split `s` into a list of substrings
- `s.upper()` – Copy of `s`; all characters converted to uppercase

String comparison

- To test for equality, use the `==` operator
- To compare order, use the `<` and `>` operators

```
user_name=raw_input("Enter your name- ")
if user_name=="Sam":
    print("Welcome back Sam")
elif user_name<"Sam":
    print("Your name is before Sam")
else:
    print("Your name is after Sam")
```

- These operators *are case sensitive*.
- Upper case characters are 'less than' lower case

Input/Output as String Manipulation

- Often we will need to do some string operations to prepare our string data for output (“pretty it up”)
- Let’s say we want to enter a date in the format “05/24/2003” and output “May 24, 2003.” How could we do that?

Input/Output as String Manipulation

- Input the date in mm/dd/yyyy format (dateStr)
- Split dateStr into month, day, and year strings
- Convert the month string into a month number
- Use the month number to lookup the month name
- Create a new date string in the form “Month Day, Year”
- Output the new date string

Input/Output as String Manipulation

- The first two lines are easily implemented!

```
dateStr = raw_input("Enter a date (mm/dd/yyyy): ")  
monthStr, dayStr, yearStr = dateStr.split("/")
```

- The date is input as a string, and then “unpacked” into the three variables by splitting it at the slashes and using simultaneous assignment.
- Next step: Convert monthStr into a number
- We can use the *int* function on monthStr to convert "05", for example, into the integer 5. (*int*("05") = 5)

Input/Output as String Manipulation

```
months = ["January", "February", ..., "December"]  
monthStr = months[int(monthStr) - 1]
```

- Remember that since we start counting at 0, we need to subtract one from the month.
- Now let's concatenate the output string together!

Input/Output as String Manipulation

```
print("The date is:", monthStr, dayStr + ",", yearStr)
```

- Notice how the comma is appended to dayStr with concatenation!

```
>>> Enter a date (mm/dd/yyyy): 01/23/2010  
The date is: January 23, 2010
```

Input/Output as String Manipulation

- Sometimes we want to convert a number into a string.
- We can use the *str* function.

```
>>> str(500)
```

```
'500'
```

```
>>> value = 3.14
```

```
>>> str(value)
```

```
'3.14'
```

```
>>> print("The value is", str(value) + ".")
```

```
The value is 3.14.
```

Input/Output as String Manipulation

- If value is a string, we can concatenate a period onto the end of it.
- If value is an int, what happens?

```
>>> value = 3.14
>>> print("The value is", value + ".")
The value is
```

```
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in -toplevel-
    print("The value is", value + ".")
TypeError: unsupported operand type(s) for +: 'float' and
'str'
```

Input/Output as String Manipulation

- We now have a complete set of type conversion operations:

Function	Meaning
<code>float(<expr>)</code>	Convert <code>expr</code> to a floating point value
<code>int(<expr>)</code>	Convert <code>expr</code> to an integer value
<code>str(<expr>)</code>	Return a string representation of <code>expr</code>
<code>eval(<string>)</code>	Evaluate string as an expression