Variables, Constants, and Data Types

- **Strings and Escape Characters**
- **Primitive Data Types**
- **Variables, Initialization, and Assignment**
- **Constants**
- **Reading for this lecture:**
  - Dawson, Chapter 2
  - http://introcs.cs.princeton.edu/python/12types

Character Strings

- So far, all of our program data has been text in *string* form. A *string* is, quite literally, a string of characters
- Test can be represented as a *string literal* by bounding it with a pair of double quotes **OR** a pair of single quotes. (Must match!)
- Examples:

```
"This is a string literal."
'123 Main Street'
'X'
"" (empty string)
```

- The word "literal" indicates that we are directly coding the information rather that getting it indirectly.

Combining Strings

- To combine (or "concatenate") two strings, we can use the plus sign

```
"Peanut butter " + "and jelly"
```

- You may find this helpful when printing output where some parts of the text may vary while other parts remain the same. Consider this example:

```
name = "Bob"
print ("Hello, " + name + "...welcome!")
```

- Prints:

```
Hello, Bob...welcome!
```

String Concatenation

- The + operator is also used for arithmetic addition
- The function that it performs depends on the type of the information on which it operates
- If both operands are strings, it performs string concatenation
- If both operands are numeric, it adds them
- `"Hello " + "world"` gives you `"Hello world"`
- `4 + 42` gives you `46`
- **NOTE:** You cannot directly concatenate a string and a number:

```
>>> print ("My favorite number is " + 7)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
>>>
```

String Concatenation

- However, it will work if you first *convert* the number to its string equivalent:

```
print ("My favorite number is " + str(7))
```

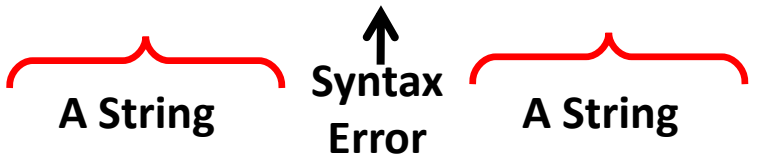
```
My favorite number is 7
```

- This has to do with the behavior of different data types in Python.
- Other programming languages create different restrictions and allowances based on how their data types are set up

Escape Sequences

- What if we want to include the quote character itself?
- The following line would confuse the interpreter because it would interpret the two pairs of quotes as two strings and the text between the strings as a syntax error:

```
print ("I said "Hello" to you.")
```



- One option would be to replace the beginning and ending double-quote symbols with single-quotes:

```
print ('I said "Hello" to you.')
```

- The *reverse* would also be valid

```
print ("I said 'Hello' to you.")
```

Escape Sequences

- Another option is to use *escape sequences*, which are character combinations that have a special meaning within a string
- Some Escape Sequences:

Escape Sequence	Meaning
`\t`	tab
`\n`	newline
`\r`	carriage return
`\"`	double quote
`\'`	single quote
`\\`	backslash

- Example:

```
print ("Hello,\n\tworld")
```

```
Hello,
   world
```

Useful *string* methods

- Using a string method requires three things:
  1) A *reference to the string*, such as a string literal or a variable
  2) The *method name*, such as **upper**
  3) The *argument list*, a pair of parentheses **()** with a list of values inside. May be empty
- Example:

```
print ("Hello")          → Hello
print ("Hello".upper())  → HELLO
print ("Hello".lower())  → hello
```

- See *Table 2.3 on page 38* of the textbook for more methods you can use

Number Bases

- You are probably used to numbers in base-10, where each digit is a 0-9 (*10 possible values*)
- The base specifies how many values can be expressed using a particular number of digits.
- For example, 3 base-10 digits can express *1000 different values*:

  `000-999`

- In other words, the **base** raised to the power the number of digits

  Example: `10^3 = 1000`

Number Bases

- In addition to base-10, you will also see other types, such as the following:
  - Binary: base-2, every digit is a `0` or 1
  - Octal: base-8, every digit is a `0`-7
  - Hexadecimal: base-16, every digit is a `0`-15; *digits `10`-15 become a-f*
- I recommend *researching* this topic to become more familiar
- In programming, you will encounter binary very frequently because that is how data is stored

Number Bases – Binary

- You are probably familiar with "bytes" as a unit of computer storage
- A byte is made of 8 bits, where each bit is a **0** or **1** – in other words, *binary*
- You have progressively larger forms of storage:
  - bits
  - bytes
  - kilobytes
  - megabytes
  - gigabytes
  - TERABYTES!!!

Types of Data

- In Python, ***all*** data are objects
- You will work mainly with two types of data:
  - **Built-in data types:**
    - These include most basic forms of data you will see in your programs
  - **Complex data types (*my wording*):**
    - Conglomerations of other data types, both built-in and other complex types
- We will introduce types as needed

Some Primitive Types

- We call these "primitive" because they form the basis for other more complex data types
- Three numeric types:

  `int`

  `float`

  `complex`
- *`True`*/*`False`* (or "boolean") values:

  `bool`
- A type for text (i.e., strings):

  `str`

Numeric Primitive Data

- The `int` type is for whole numbers:

  `7, -358, 0, -10, 12398`

- The `float` type is for decimal (or "floating-point") numbers:

  `7.6, -35.8, 0.0, -1.09, 12398.0`

- The `complex` type is for numbers with an imaginary component. (We *probably* will not use this type.)

- Each of these will have different behaviors and limitations, depending on a number of factors

Boolean Primitive Data

- A `bool` type can have either of two values:

  `True`

  `False`

- `True` and `False` are reserved words in Python
- A `bool` type can be useful for representing any two states such as a light bulb being on or off

```
on = True
```

`String (str)` Data

- As mentioned earlier, a "string" is a sequence of ***zero or more*** characters
- You will use strings often, in different ways:
  - Printing as output
  - Fetching as input
  - Comparing
  - Reversing
  - Converting to/from other types
- Work and practice to become comfortable with this type and its many uses

Characters

- Some languages, such as Java, have a character type, specifically
- Python does not, though, and if you need to use a character, you will likely just use a *string* consisting of a single character
- Each character, however, will correspond to an *integer value* in some *character set*, and there are methods to perform conversions:
  - Integer to character: `chr`
  - Example: `chr(97)` → `a`
  - Character to integer: `ord`
  - Example: `ord('a')` → `97`

Character Sets

- A *character set* is an ordered list of characters, with each character corresponding to a unique number
- Python uses the *Unicode character set*
- The Unicode character set uses sixteen bits per character, allowing for **65,536 (2^16)** unique characters
- It is an international character set, containing symbols and characters from many world languages

Characters

- The *ASCII character set* is older and smaller (***8-bit***) than Unicode, but is still quite popular (in C programs)
- The ASCII characters are a subset of the Unicode character set, including:

uppercase letters	**A, B, C, …**
lowercase letters	**a, b, c, …**
punctuation	**period, semi-colon, …**
digits	**0, 1, 2, …**
special symbols	**&, |, \, …**
control characters	**carriage return, tab, ...**

Variable Declaration

- A *variable* is a name for a location in memory
- A variable must be *declared* by specifying its name and its initial value

```
name = "Bob"
body_temp = 98.6
light_on = False
```

- In some languages (*e.g., Java*), variables are of a specific type, but Python is more flexible

Constants

- A constant is an identifier that is similar to a variable except that it is meant to *hold the same value during its entire existence*
- As the name implies, it is constant, not variable
- In Python, we indicate a constant using ALL CAPS

```
MIN_HEIGHT = 69
```

- This indicates that the value should not be changed after it is first declared
- Some programming languages will actually forbit you to change the value of a constant

Constants

- Constants are useful for three important reasons
- First, they give meaning to otherwise unclear literal values
  - For example, **NUM_STATES** is more meaningful than the literal 50
- Second, they facilitate program maintenance
  - If a constant is used in multiple places and you need to change its value later, its value needs to be updated in only one place – what if the country gets a **51st** state?
  - Rather than having to find and change it in multiple places!
- Third, they formally show that a value should not change, avoiding inadvertent errors by other programmers

Value Assignment

- An *assignment statement* gives the variable an actual value in memory
- The equals sign provides this function

```
total = 55
```

- The expression on the right is evaluated and the result is stored as the value of the variable on the left
- Any value previously stored in **`total`** is overwritten
  - Unlike some other languages, Python allows you to store any type of data in any variable.
- Other languages - like Java – will restricted the kinds of values you can assign to a variable, based on its type

Variables and Literals

- i = 7
- j = -8.7
- k = 9
- c = “Hello World"
- is_it_on = True