**Lecture #2  Python Data Type and Variables**

**Introduction**

In terms of programming, a “**data**” is an individual object that represents a qualitative or a quantitative value, such as a color or 23 units of a product. In order for the Python interpreter to implement an object to comply with the IPO (input, processing, and output) model of a computing, the object must be represented by an appropriate type of **data**.

A girl’s first name, such as “Taylor”, is an intangible object which is represented by a combination of English characters: T-a-y-l-o-r. Since a girl’s first name can only consist of alphabets (no numerals, no symbols), it is necessary to define rules for picking only the proper characters to form a data that represents of a girl’s first name. With the rules, no one can enter a combination of characters like 452 (obviously a numerical value) to a data field that stores data of a girl’s first name. Similarly, a data field that holds girl’s age should not be assigned a value like “Alicia”.

The term “**data type**” (or simply “type”) refers to the definition of nature settings of a Python object in order to properly represent the object and process it with Python’s language core. In computing, a value that carries a meaning of some kind is known as a “**literal**”. A **string literal** is a sequence of characters enclosed by either a pair of single quotes or a pair of double quotes. A string literal represents a string values. A **number** is a sequence of digits that are not enclosed by quotes. A number represents a numerical value in units. In Python, the most frequently used types are data of numbers and string values.

As discussed in a previous lecture, the “data type” is implicitly determined by the Python interpreter as the moment when a value is to assign to a variable. This lecture, thus, will start with the discussion of available Python data types.

**Over of Python data types**

Python’s language core comes with a set pre-defined data types, known as “**primitive data type**”. They are the built-in data types encapsulated in several “object classes” of Python. Each object class has a unique identifier, such as “int”, “float”, “str”, “bool”, and “NoneType”. The term “to be encapsulated” in programming means “to be defined”.

Similar types of data are grouped in categories. The following table provides an overview of available categories of Python’s primitive data types.

<table>
<thead>
<tr>
<th>Categories</th>
<th>Description</th>
<th>Python class</th>
</tr>
</thead>
<tbody>
<tr>
<td>numbers</td>
<td>Numbers are typically made of numerical characters and the decimal sign (.). Some numbers are expressed in scientific notation, such as 2.3e25 (which means 2.3×10^{25}).</td>
<td>int and float</td>
</tr>
<tr>
<td>string</td>
<td>A string is a combination of any Python recognizable characters (even in other natural language like Chinese, Japanese, and Farsi).</td>
<td>str</td>
</tr>
<tr>
<td>Boolean</td>
<td>Boolean value are strictly limited to two possible values: True and False. Sometime, True is represented by 1 while False is represented by 0.</td>
<td>bool</td>
</tr>
<tr>
<td>container</td>
<td>The “container” type refers to a group of data types that collect Python objects in a “container” object.</td>
<td>tuple, list, dictionary</td>
</tr>
</tbody>
</table>
A Python set is an unordered collection of elements. It is used for calculation based on the Set Theory of mathematics.

The keyword `None` is a single value that signify “absence of a value”.

The next several sections will start with the discussion about what a variable is in Python, and continue to discuss each of these categories in detail.

### Concept of variables

In programming, the term “variable” refers to a programmer-defined name that represents a group of registers in the physical memory (typically the DRAMs). These registers make up a fixed number of bits, as specified by the data type, to temporarily host the value that is assigned to the variable. Since the value stored in these registers can be modified, they are “variable” values. In the following statement, “x” is the identifier of a variable that is assigned 4265 as value. In other words, a Python value “x” is declared to hold an integer 4265.

```python
>>> x = 4265
```

The following figure illustrates how a series of consecutive registers in a DRAM, from 21st to the 24th, are grouped to keep an integer 4265. In this scenario, “x” is the identifier of the variable and is also the temporarily name of the group of registers.

<table>
<thead>
<tr>
<th>Register</th>
<th>identifier</th>
<th>The value it holds</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>x</td>
<td>4265</td>
</tr>
<tr>
<td>22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Python programmers can assign an identifier to a Python object through the declaration of a variable. The following illustrates how to: (a) designate an identifier to a variable; (b) assign a value to the variable; and (c) set the data type of the variable.

```python
>>> x = 2.1
>>> type(x)
<class 'float'>
>>> id(x)
20027688
>>> y = 'orange'
>>> type(y)
<class 'str'>
```

In the above example, the statement `x = 2.1` reads “assign a number 2.1 to the variable x” and the statement `y = 'orange'` reads “assign a string literal ‘orange’ to y”. The equal sign (=) represents the “assignment” operator which assign a value to a variable. A later lecture will discuss about Python operators in details. By the way, in terms of Python, the value 2.1 in the statement `x = 2.1` and the value “orange” of the `y = ‘orange’` statements are referred to a “literal”.

Variables are useful in problem solving, particularly when some mathematical calculation is required. In Physics, the concept of “uniform motion” is described as \( x = vt \), where \( x \) is the distance traveled, \( v \) is the constant velocity, and \( t \) is the time the object is in motion. The equation, \( x = vt \), uses three variables: \( x \), \( v \), and \( t \). A Python programmer can write a program to solve the following problem.
“Jennifer drives along the freeway to Las Vegas. She sets the cruise on 80 mile per hour. After 30 minutes, how far has she travelled?”

The following is a sequential list of statements to solve this problem in Python. The answer is 40.0 miles.

```python
>>> v = 80
>>> t = 0.5
>>> x = v*t
>>> x
40.0
```

In terms of object-oriented programming, officially designate a meaningful identifier to a variable is known as “declaration”. Declaring a variable provides a way to label and access data without the need to know exactly where in the physical memory the data is stored. However, unlike other language, Python does not require the programmer to specify the data type when creating a variable. Python interpreter will automatically regulate the data type of a variable according to the value it is assigned to host. In the following, the variable \( x \) is set to be “int” type first, then “str” and “tuple” accordingly.

```python
>>> x = 2
>>> type(2)  # int
<class 'int'>
>>> x = "Python"
>>> type(x)  # str
<class 'str'>
>>> x = (9, 8, 7, 6)
>>> type(x)  # tuple
<class 'tuple'>
```

The following is another example that demonstrates the feasibility to use one variable name to represent different kinds of objects at different times.

```python
>>> x = 0  # x bound to an integer object
>>> x = "Hello"  # Now it's a string.
>>> x = [1, 2, 3]  # And now it's a list.
```

As illustrated in a previous lecture, another way to display the value stored in a variable is to use the `print()` function, as shown below.

```python
>>> firstname = "Larry"
>>> print(firstname)
'Larry'
```

The above statements create a variable called “firstname” and assign it with the value “Larry”. The `print()` function then displays the value assigned to firstname on the screen.

Python recommends a naming convention for creating variable identifiers. Basically, a variable identifier (or variable name) must: (a) start with either an underscore (_) or a letter (such as “a” or “A”); (b) followed by a combination of alphanumerical characters (letters and digits), except the blank space; (c) use the underscore (_) to con; and (d) avoid using any Python reserved keywords. The following table lists some of the reserved keywords.

<table>
<thead>
<tr>
<th>Table: Python reserved keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>and</code> <code>del</code> <code>for</code> <code>is</code> <code>raise</code></td>
</tr>
<tr>
<td><code>assert</code> <code>elif</code> <code>from</code> <code>lambda</code> <code>return</code></td>
</tr>
<tr>
<td><code>break</code> <code>else</code> <code>global</code> <code>not</code> <code>try</code></td>
</tr>
<tr>
<td><code>class</code> <code>except</code> <code>if</code> <code>or</code> <code>while</code></td>
</tr>
</tbody>
</table>
Under this naming convention, the following are examples of legal variable identifiers.
However, `class_1_Spam, spam$, and my@@!` are illegal variable names. Also, Python is case-
sensitive, `SPAM, spam, SpAm, sPAm, and SpAM` are different identifiers.

```python
>>> first_name = "Taylor"
>>> _last_name = "Swift"
>>> _age = 24
>>> Best_song = "You Belong With Me"
>>> year = 2009
```

It is also necessary to note that identifiers that have two leading and trailing underscores, such as `__main__`, are system-defined names. They have special meanings to the interpreter; therefore, programmers should avoid using two leading or trailing underscore (`_`) to name Python variables. A later lecture will discuss about such identifiers in detail.

Numerical types

In the “numbers” category, Python 2 data could be one of the four types: `int`, `long`, `float`, and `complex`. Interestingly, Python 3 drop the `long` type and preserves only three distinct numeric types: integers, floating point numbers, and complex numbers.

Numerical data are values that represent a number of units. A numerical literal consists of digits only (no letters, not symbols) and is used to express a number of units. They should not be enclosed by quotes. Python 3 provide three object class to define numerical data: `int` (short for integer), `float` (short for floating-point), and `complex` (short for complex numbers). The following is a complete list of numerical types supported by Python 3.

**Table: Numeric literals**

<table>
<thead>
<tr>
<th>Type</th>
<th>Mathematic name</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>Integers</td>
<td>&gt;&gt;&gt; 1234</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1234</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&gt;&gt;&gt; -24</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-24</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&gt;&gt;&gt; 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>float</td>
<td>Floating-point</td>
<td>&gt;&gt;&gt; 1.23</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&gt;&gt;&gt; 3.14e-10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&gt;&gt;&gt; 5.07E210</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&gt;&gt;&gt; 5.07e+210</td>
</tr>
<tr>
<td>complex</td>
<td>Complex number</td>
<td>3+4j, 3.0+4.0j, 3j</td>
</tr>
</tbody>
</table>

Integers are whole number that can be positive, negative, or 0 and is written in a form without comma(s). In Python, the `int` type can be expressed in decimal format (the base is 10), which is the default format. The following prints out a negative integer expressed in the decimal format.

```python
>>> print(-25)
-25
```

In Python, an integer can also be expressed in binary (the base is 2), octal (the base is 8), or hexadecimal (the base is 16) formats. Except the decimal format, all other formats required a prefix to specify their formats. The following table lists the formats and their prefixes with examples. By the way, prefixes are not case-sensitive.

**Table: Formats**

<table>
<thead>
<tr>
<th>Format</th>
<th>Prefix</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Decimal | N/A | >>> 124
---|---|---
Binary | 0b (or 0B) | >>> 0b1111100
Octal | 0o (or 0O) | >>> 0o174
Hexadecimal | 0x (or 0X) | >>> 0x7c

Binary literals start with a leading prefix zero-b, “0b” (or “0B”), followed by a series of 0’s and 1’s. The following demonstrates how to specify a value in binary format. The value 124 in decimal format is equivalent to 1111100 in binary, 174 in octal, and 7c in hexadecimal.

```python
>>> 0b1111100
124
```

The following demonstrates how Python interpreter automatically convert non-decimal literals to their decimal equivalents and display only the decimal values as results. Apparently, the conversion is based on the formula: \[\sum a_i \times b^{i-1} = a_1 \times b^0 + a_2 \times b^1 + a_3 \times b^2 + \ldots + a_n \times b^{n-1},\]
where \(a_n\) is the \(n\)-th digit (counting from the right to left) and \(b\) is the base.

```python
>>> 1, 10, 100        # decimal literals
(1, 10, 100)
>>> 0b1, 0b10, 0b100  # binary literals
(1, 2, 4)
>>> 0o1, 0o10, 0o100  # Octal literals
(1, 8, 64)
>>> 0x01, 0x10, 0xFF   # Hex literals
(1, 16, 255)
```

The binary literal, 0b1010, is calculated by the Python interpreter with the base being 2 to find its decimal equivalent: \(1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 11\).

```python
>>> 0b1010
10
```

Octal literals start with a leading prefix zero-o, 0o (or 0O), followed by a series of octal digits 0~7; hexadecimals start with a leading prefix 0x (or 0X), followed by hexadecimal digits 0~9, and A~F. In hexadecimal, “A”, “B”, “C”, “D”, “E”, and “F” (or “a”, “b”, “c”, “d”, “e”, and “f”), as digits that represent 10, 11, 12, 13, 14, and 15 respectively. Hexadecimal digits are not case sensitive.

```python
>>> 0xFF == 0xff == 0XFf == 0xFf
True
```

A number express in 0o37 = \(3 \times 8^1 + 7 \times 8^0 = 31\).

```python
>>> y = 0o37
31
```

The octal literal, 0o177, is calculated with the base being 8: \(1 \times 8^2 + 7 \times 8^1 + 7 \times 8^0 = 127\)

```python
>>> 0o177
127
```

The hexadecimal literal, 0x9ff, is calculated with the base being 16: \(9 \times 16^2 + f \times 16^1 + f \times 16^0 = 9 \times 16^2 + 15 \times 16^1 + 15 \times 16^0 = 2559\).

```python
>>> 0x9ff
2559
```
A number expressed in 0xA0F = $A \times 16^2 + 0 \times 16^1 + F \times 16^0 = 10 \times 16^2 + 0 \times 16^1 + 15 \times 16^0 = 2575$.

```python
>>> x = 0xA0F
2575
```

All the above numbers are created by specifying numeric literals. Yet, Python numbers can also be created as the result of built-in functions and operators. The following use the “division” operator to produce a number of float type.

```python
>>> print(5.0/9.0)
0.5555555555555556
```

The following use the “`bin()`” function to convert a number 1234 to its binary format.

```python
>>> bin(1234)
'0b10011010010'
```

Although Python prints in decimal by default, Python also provides few built-in functions to convert decimal integers to their octal and hexadecimal equivalents. The `oct()` function converts decimal to octal, and the `hex()` to hexadecimal.

```python
>>> oct(64), hex(64), hex(255)
('0o100', '0x40', '0xff')
```

The `int()` function can also convert a “string of digits” to an integer with an optional second argument that specifies the numeric base. A “string of digits” means a combination of digits that is enclosed by a pair of quotes. If the second argument is absent, the base is assumed to be 10.

```python
>>> int('0100'), int('0100', 8), int('0x40', 16)
(100, 64, 64)
```

The `eval()` function treats strings as though they were Python code. It therefore has a similar effect:

```python
>>> eval('100'), eval('0o100'), eval('0x40')
(100, 64, 64)
```

Python also supports long integers, yet the instructor could not find any definition of range to specify the maximum and minimum. The following are three very large integers. By the way, two asterisk signs (**) represents the exponentiation operator. The statement $3 \times 2$ reads “3 raised to the second power”.

```python
>>> 999999999999999999999999999 + 1
1000000000000000000000000000000
>>> (123456789 * 987654321) ** 2
14867566530049990397812181822702361
>>> -12345678909876543211234567890987654321
-12345678909876543211234567890987654321
```

The term “**floating-point number**” refers any number that can contain a fractional part next to the decimal point, such as 3.14, -6.1723, and 18238.04956321697. In other words, floating point numbers are numerical values with fractional parts, such as 53.16148.
As stated in a previous lecture, the precision of calculation result varies from one operating systems (OS) to another. This is not a bug or error of the operation system. It is a result of the fact that most decimal fractions cannot be represented exactly as a float. Floating-point numbers are represented in computer hardware as binary fractions (base 2), yet, they must be converted to decimal fractions (base 10). Unfortunately, the result of conversion is usually a close approximate, not the exact value. The following compares the outputs between Windows 7/8 and Linux. The statement simply asks the Python prompt to display 0.3. The result of Windows 7/8 is much closer to human readers’ expectation.

<table>
<thead>
<tr>
<th>Windows 7/8</th>
<th>Linux</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&gt;&gt;&gt; 0.3</code></td>
<td></td>
</tr>
<tr>
<td><code>0.3</code></td>
<td><code>0.2999999999999999</code></td>
</tr>
</tbody>
</table>

One way to resolve the precision issue is to use the `format()` function. The directive, “.1f”, specifies that only 1 digit next to the decimal point.

```
>>> format(0.3, "0.3")
'0.3'
```

In Python, float literals may be expressed in scientific notation, with `E` or `e` indicating the power of 10. The number, 4.17×10⁻²⁵, can be expressed as either `4.17E-25` or `4.17e-25`, as proved below. 6.25e-23 is the scientific form of 6.25×10⁻²³, while 4.173e15 is 4.173×10¹⁵.

```
>>> 4.17E25 == 4.17e25
True
>>> 6.25e-23
6.25e-23
>>> 4.173e15
4.173e15
```

In Python, two equal signs (==) represents the equality operator which checks if two literals are equivalent. The returned value, True, indicates that the two sides of literal are the same. As specified in scientific notation, the positive sign (+) of an exponent is optional.

```
>>> 5.07e+210 == 5.07e210
True
>>> 5.07e+210 == 5.07E210
True
```

By the way, the instructor was not able to find reliable reference that can provide the limit of floating-point values.

```
>>> 4.5e219 * 3.1e14
1.395e+234
```

Numerical values can be either positive or negative. The sign of an integer is denoted by `+` or `-` for positive or negative. The positive sign is default; therefore, any numerical value with a negative sign (-) is assumed to be positive. In some operating systems, -5.19e24 might be displayed as -5.18999999999999e+24.

```
>>> -51
-51
>>> -3.173
-3.173
>>> -5.19e24
-5.19e24
```

Any number, when multiplied with -1, change its sign.
In mathematics, a complex number is an ordered pair of real floating-point numbers denoted by \( a + bj \), where \( a \) is the real part and \( b \) is the imaginary part of the complex number. Python support complex data type to define a complex literal, as shown below, in which 3 is a real number, while 5j is an imaginary value. The imaginary part must be suffixed by \( a \) \( j \) or \( J \).

\[
>>> 3 + 5j
\]

The following is another sample of complex number.

\[
>>> 3 + 26J
\]

A complex number with a real part of 2 and an imaginary part of -3 is written as \( 2 + -3j \) or \( 2 - 3j \).

\[
>>> (2 + -3j) == (2 - 3j)
True
\]

When the real part is 0, a complex number can be simplified to \( nj \) or \( nJ \).

\[
>>> (0 + 7J) == 7J
True
>>> (0 - 4j) == -4j
True
\]

The followings are some examples of complex math at work:

\[
>>> 1j * 1J
(-1+0j)
\]

\[
>>> 2 + 1j * 3
(2+3j)
\]

\[
>>> (2+1j)*3
(6+3j)
\]

The following demonstrates how to assign a numerical value to a variable. In some operating systems, 37.4 might be displayed as 37.399999999999999.

\[
>>> x = 12
>>> y = 37.4
>>> z = 9.1e6
>>> x, y, z
(12, 37.4, 9100000.0)
>>> x = (3 + 5j)
>>> y = (2 - 1j)
>>> x, y
((3+5j), (2-1j))
\]

Variables of numerical type can undergo arithmetic operations, such as addition (+), subtraction (-), multiplication (*), and division (/).

\[
>>> x = 12
>>> y = 37.4
>>> z = 9.1e6
>>> x + y + z
9100049.4
>>> x - y - z
-9100025.4
\]
>>> x * y * z
4084079999.9999995
>>> x / y / z
3.515885878827055e-08
>>> (3 + 5j) * (2 - 1j)
(11 + 7j)

By the way, the math of complex is similar to the math of integers.

\[
(3 + 5j) \times (2 - 1j)
= 3 \times 2 + 3 \times (-1j) + 5j \times 2 + 5j \times (-1j)
= 6 -3j + 10j - 5j^2
= 6 + 7j + 5 \times (-1)
= 11 + 7j
\]

**String types**

A string in Python is defined as a combination of characters enclosed by a pair of double quotes or a pair of single quotes. In other words, a string is a series of individual characters (including the blank space) enclosed by a pair of quotes.

```python
>>> 'S1562'  # single quotes
'S1562'
>>> "Classic Jazz music"  # double quotes
'Classic Jazz music'
```

Since a string literal must be enclosed by either a pair of single quotes or a pair of double quotes, when a literal is enclosed by quotes, the value must be a string type. The following is a list of examples of string literals, although they could look like a number to a human reader.

```python
>>> "16.9"
16.9
>>> '37'
37
>>> "Python"
'Python'
>>> 'programming'
'programming'
```

Interestingly, in addition to single and double quotes, triple quotes similar to work as well. A pair of triple quotes consists of either three consecutive single quotes ("""") or three double quotes ("""""""""). The following demonstrates the use of triple quotes to define string literals.

```python
>>> """Recycle the trash to save the earth"""
'Recycle the trash to save the earth'
>>> """Taipei 101"""
'Taipei 101'
```

As a matter of fact, triple-quoted strings capture all the text that appears prior to the terminating triple quote, as opposed to single- and double-quoted strings, which must be specified on one logical line. Therefore, triple-quoted strings are useful when the contents of a string literal span multiple lines of text such as the following.

```python
>>> print(""""Content-type: text/html
... <h1> Hello World </h1>
... Click <a href="http://www.python.org">here</a>.
... """")
Content-type: text/html
<h1> Hello World </h1>
Click <a href="http://www.python.org">here</a>.
```
Two or more string literals can be concatenated with concatenation operator (+) to make a new single string. In the following, the instructors demonstrates how to combine three string literals (the blank space is counted as one string literal).

```python
>>> "Python" + " " + "programming"
'Python programming'
```

Another way to concatenate two or more string literals is to use the `print()` function with comma (,) as delimiter.

```python
>>> print("Python", "3.4", "programming")
Python 3.4 programming
```

Interestingly, Python does not provide any data type that is similar to the “char” type of C or C++. Therefore, the statement ‘a’ and "a" both are treated as a single-character string by the Python interpreter.

```python
>>> type('a')
<class 'str'>
>>> type("a")
<class 'str'>
```

Since a string is a sequential combination of individual characters within a pair of quotes, a Python string can be treated as an collection of sequentially indexed characters. In the following, each character in the string literal “happy hours”, including the blank space, has an integer index. The “size” of the string refers to the total number of character the string has. Python also refers the term “size” to “length”. The `len()` function can return the size of the string. The first index is 0, the last index is size-1.

```python
>>> s = "happy hours"
>>> len(s)
11
>>> s[0], s[1], s[2], s[3], s[4], s[5], s[6], s[7], s[8], s[9], s[10]
('h', 'a', 'p', 'p', 'y', ' ', 'h', 'o', 'u', 'r', 's')
```

In the above example, the instructor assigned the string literal “happy hours” to a variable named “s”. This string literal consists of 11 characters, including the blank space. Each character of the string can thus be represented as s[i], where i is the index in a range from 0 to size-1. Apparently, programmers can extract a single character with the format s[i] from a string literal. For instance, s[0] is “h” and s[4] is “y”.

```python
>>> s = "happy hours"
>>> s[4]
y
```

Since `len(s)` return the “size” of s, the index can be counted backwards. The index of the last letter is `len(s) - 1`, the second last is `len(s) - 2`; therefore, `s[len(s) - 3]`, for example, will return the letter “u”, as shown below.

```python
>>> s = "happy hours"
>>> s[len(s) - 3]
u
```

With the range operator (:), programmers can extract a substring, defined by `s[i:j]`. The “substring” consists all characters in “s” whose index k is in the range i <= k < j. If either i or j index is omitted, the beginning or end of the string is assumed, respectively.
>>> s[0:5]
'happy'
>>> s[2:5]
'ppy'
>>> s[:5]
'happy'
>>> s[6:]
'hours'

Other data types can be converted into a string by using either the `str()` or `repr()` function. In many cases, `str()` and `repr()` will return identical results. However, they have subtle differences in semantics that will be described in a later lecture.

```python
>>> y = 63.5
>>> type(y)
<class 'float'>
>>> type(str(y))
<class 'str'>
>>> type(str(3.14))
<class 'str'>
>>> type(repr(7.12))
<class 'str'>
```

The following is another demonstration.

```python
>>> 'Python programming ' + str(3) + " units"
'Python programming 3 units'
>>> 'Python programming ' + repr(3) + " units"
'Python programming 3 units'
```

A later lecture will discuss about these functions in details. In the following statements, the instructor uses both `upper()` and `lower()` functions to demonstrates how Python “string functions” can implement string literals. The `upper()` function returns the uppercase version of the string, while the `lower()` function returns the lowercase version of the string. However, both functions only temporarily change the way the string literals looks, they do not actually modify the string literals.

```python
>>> str = "Welcome To Python!"
>>> print(str.upper())
WELCOME TO PYTHON!
>>> print(str.lower())
welcome to python!
```

The “bool” type

The “bool” type of data is an implementation of conditional expressions that returns a Boolean value as result. A Boolean value can only be one of the two possible values: True or False. Internally, at the runtime, True is represented as 1 and False as 0, and they can be used in numeric expressions as those values.

```python
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

The following is an example of a Boolean statement.

```python
>>> 3 > 2
True
```

The following is another example. It is necessary to note that only the `x < 4` statement is a Boolean statement, because the `x = 5` statement simply assigns a value to `x`. 

Python Programming – Penniel P. Wu, PhD. 50
Typically, a “bool” data is a variable that stores and returns a Boolean value. In the following example, y stores the result of a conditional expression x < 4; therefore, y is a variable of the “bool” type, because the statement (x < 4) produces a Boolean value for y to hold. In the above example, x holds a integer, it is a variable of “int” type.

```
>>> x = 5
>>> x < 4
False
>>> type(x)
<class 'int'>
```

```
>>> x = 5
>>> y = (x < 4)
>>> type(y)
<class 'bool'>
```

Many Python functions return a Boolean value. The `isalpha()` function, for instance, can check if all char in the string are alphabetic.

```
>>> s = "D901"
>>> s.isalpha()
False
```

Decisive structures, such as the `if` structure, requires one or more conditional expressions that return a Boolean value. A later lecture will discuss about decisive structure in details. If the conditional expression (x % 2 == 0) return True, display “even number”; otherwise, display “odd number”.

```
>>> x = 7
>>> if (x % 2 == 0):
...    print("even number")
...  else:
...    print("odd number")
...  
odd number
```

Repetition structure, such as the while loop, might also use conditional expressions that return a Boolean value. In the following example, the conditional expression (i < 10) is evaluated during every iteration. If the returned value is True, the while loop stops. A later lecture will discuss about repetition structures in details.

```
>>> i = 0
>>> while (i < 10):
...    print(i*i)
...    i = i + 1
...
0
1
4
9
16
25
36
49
64
81
```

Python container types

Python has two generic container type objects, the `sequence` type and the `mapping` type. All the container types have two things in common: (a) all of their elements are enclosed by a pair
of “container symbols”, yet tuples could have exceptions; and (b) all elements can be identified by the assigned index (sequence) or key (mapping) in the format of \textit{objectID[index]} or \textit{objectID[key]}.

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Category</th>
<th>Symbols</th>
<th>Example</th>
</tr>
</thead>
</table>
| list | list | Sequence (mutable) | [] | >>> l1 = [3, 6, 9]  
>>> l2 = ['A', 'B', 'C', 'D']  
>>> l3 = ['Python', 3.4, True] |
| tuple | tuple | Sequence (immutable) | () | >>> t1 = (3, 6, 9)  
>>> t2 = ('A', 'B', 'C', 'D')  
>>> t3 = ('Python', 3.4, True) |
| dict | dictionary | Mapping (mutable) | {} | >>> d={'OH':'Ohio', 'TX':'Texas'} |

A \textit{sequence} is an ordered collection of objects. “Sequence” objects are created to store multiple values in an organized and efficient fashion. Each element in a “sequence” object is an individual literal and is designed an index which specifies the element’s sequence. In addition to the “\textit{str}” type, Python produce a few more data types of the “sequence” category: “\textit{list}” and “\textit{tuple}”. Python lists are constructed with square brackets. All its elements are separated by commas. However, elements in a list do not have to be the same types. The following are two lists. The first lists contains three elements of “\textit{str}” types, the second contains four elements of different types. During the construction period, Python interpreter automatically assign the index to each element. The first element is assigned 0 as index.

\begin{verbatim}
>>> ['a', 'b', 'c']
['a', 'b', 'c']
>>> [1, 'apple', True, 6.3]
[1, 'apple', True, 6.3]
\end{verbatim}

To better implement Python lists, programmer frequently assign them to a variable. The variable in turn serves as an identifier of the list.

\begin{verbatim}
>>> x = ['a', 'b', 'c']
>>> type(x)
<class 'list'>
>>> y = [1, 'apple', True, 6.3]
>>> type(y)
<class 'list'>
\end{verbatim}

With an identifier, each element of the list can be identified and its values can be retrieved using the \textit{objectID[index]} format.

\begin{verbatim}
>>> x = ['a', 'b', 'c']
>>> x[0]
'a'
>>> x[1]
'b'
>>> x[2]
'c'
\end{verbatim}

The \texttt{len()} function can return the total number of element in a list. The number is known as the “size” or “length” of the list.

\begin{verbatim}
>>> y = [1, 'apple', True, 6.3]
\end{verbatim}
>>> len(y)
4
>>> len(['a', 'b', 'c'])
3

Values of an element of a list is mutable, which means they can be re-assigned with different value. In the following, the instructor use the `x[2] = 'd'` statement to change the value of `x[2]` from `c` to `d`.

```python
>>> x = ['a', 'b', 'c']
>>> x
['a', 'b', 'c']
>>> x[2] = 'd'
>>> x
['a', 'b', 'd']
```

By the same token, the value of `y[1]` in the following example is changed from “apple” to “orange”.

```python
>>> y = [1, 'apple', True, 6.3]
>>> y[1] = "orange"
>>> y
[1, 'orange', True, 6.3]
```

Python tuples are constructed with or without enclosing parentheses. Elements in a tuples are separated by commas.

```python
>>> ('F', 'B', 'I')
('F', 'B', 'I')
>>> 'C', 'I', 'A'
('C', 'I', 'A')
```

Elements of a tuple do not have to be the same type. The following tuple is made of elements of ‘int’, ‘str’, and ‘bool’ types.

```python
>>> (1, 'a', 'cherry', True, 7.39)
```

Assigning a tuple to a variable is an effective way to implement tuples. In the following example, “course” is the identifier of a Python tuple. The instructor then displays value of the `course[1]` element. The `len()` function can return the “size” or “length” of the tuple.

```python
>>> course = ('Perl', 247, "PHP")
>>> course
('Perl', 247, "PHP")
>>> course[1]
247
>>> len(course)
3
```

It is necessary to note that Python tuples are immutable, which means the value of its elements cannot be modified after the tuple is successfully constructed. The statement `course[1] = "Python"` is illegal, and Python interpreter will return an error message: 'tuple' object does not support item assignment.

A mapping object uses immutable (unchangeable) values, as user-specified keys, to map another objects. As of Python 3, there is only one standard mapping type known as the “dict” (short for “dictionary”) type. A non-empty Python dictionary contains one or more “key-value pairs”. Each “key-value pairs” consists of a key and a value in the format of `key:value`. 
Assigning a dictionary to a variable could be an effective way to implement it. In the following example, the instructor uses the statement \texttt{month[1] = 'January'} to change the value of the \texttt{month[1]} element. The \texttt{len()} function can return the “size” or “length” of the dictionary. By the way, Python dictionaries are mutable.

\begin{verbatim}
>>> month = {1:'Jan', 2:'Feb', 3:'Mar'}
>>> month[1] = 'January'
>>> month
{1: 'January', 2: 'Feb', 3: 'Mar'}
\end{verbatim}

Interestingly, both \texttt{key} and \texttt{value} can be any Python data type. In the following, the instructor creates several Python dictionaries to demonstrate a variety of data types.

\begin{verbatim}
>>> airport = {'PHL':'Philadelphia', 'PIT':'Pittsburgh'}
>>> gender = {'F':'Female', 'M':'Male'}
>>> areaCode = {714:'Anaheim', 818:'Burbank', 949:'Irvine'}
>>> abbr = { 5:500, 7:700, 9:900 }
>>> tax = { '8.4%':0.084, '8.5%':0.085 }
>>> discount = { 0.9:'10\% off', 0.8:'20\% off', 0.7:'30\% off' }
\end{verbatim}

The following table compares the data type of keys and values used in the above dictionaries. It also illustrates how to retrieve data from elements of each dictionary.

<table>
<thead>
<tr>
<th>Dictionary</th>
<th>Data type</th>
<th>Data Retrieval</th>
</tr>
</thead>
<tbody>
<tr>
<td>airport</td>
<td>str, str</td>
<td>&gt;&gt;&gt; airport['PIT'] 'Pittsburgh'</td>
</tr>
<tr>
<td>gender</td>
<td>str, str</td>
<td>&gt;&gt;&gt; gender['F'] 'Female'</td>
</tr>
<tr>
<td>areaCode</td>
<td>int, str</td>
<td>&gt;&gt;&gt; areaCode[818] 'Burbank'</td>
</tr>
<tr>
<td>abbr</td>
<td>int, int</td>
<td>&gt;&gt;&gt; abbr[9] 900</td>
</tr>
<tr>
<td>IsLeap</td>
<td>int, bool</td>
<td>&gt;&gt;&gt; IsLeap[2008] True</td>
</tr>
<tr>
<td>tax</td>
<td>str, float</td>
<td>&gt;&gt;&gt; tax['8.5%'] 0.085</td>
</tr>
<tr>
<td>discount</td>
<td>float, str</td>
<td>&gt;&gt;&gt; discount[0.8] '20% off'</td>
</tr>
</tbody>
</table>

The following table illustrates how to create empty “container” objects. The word “empty” describes a situation that a list, a tuple, or a dictionary is created without any element.

<table>
<thead>
<tr>
<th>Object type</th>
<th>Without variable</th>
<th>With variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empty string</td>
<td>&gt;&gt;&gt; &quot;&quot; (or &gt;&gt;&gt; '')</td>
<td>&gt;&gt;&gt; s = &quot;&quot;</td>
</tr>
<tr>
<td>Empty tuple</td>
<td>&gt;&gt;&gt; ()</td>
<td>&gt;&gt;&gt; t = ()</td>
</tr>
<tr>
<td>Empty list</td>
<td>&gt;&gt;&gt; []</td>
<td>&gt;&gt;&gt; l = []</td>
</tr>
<tr>
<td>Empty dictionary</td>
<td>&gt;&gt;&gt; {}</td>
<td>&gt;&gt;&gt; d = {}</td>
</tr>
</tbody>
</table>

The “container” type of data are very useful in programming, especially those involved in data manipulations. The instructor will discuss the implementation of the “container” types in details in a later lecture. The following is a demonstration of how the “container” types help to manipulate data based on a mathematic theory.
Both string and tuple are not mutable, an empty string or an empty tuple cannot be inserted any element. However, lists and dictionaries are mutable. Elements can be inserted to an empty list or an empty dictionary. In the following, the instructor demonstrates how Python lists works with the `append()` function to add new elements to a list.

```python
>>> l = [] # empty list
>>> l.append('a')
>>> l.append('b')
>>> l.append('c')
>>> l
['a', 'b', 'c']
```

In the following, the instructor demonstrates how to add new elements to a dictionary with the `objectID[key] = value` syntax.

```python
>>> d = {} # empty dictionary
>>> d['OH'] = "Ohio"
>>> d['CA'] = "California"
>>> d['PA'] = "Pennsylvania"
>>> d['MI'] = "Michigan"
>>> d
{'CA': 'California', 'PA': 'Pennsylvania', 'MI': 'Michigan', 'OH': 'Ohio'}
```

The `sets` module provides classes for constructing and manipulating unordered collections. With this module, programmers can create a Python set which is an unordered collection of elements. Elements of a set can be populated as a Python “list”, “tuple”, or “dictionary”, as demonstrated below.

```python
>>> x = set([1, 2, 3])
>>> type(x)
<class 'set'>
>>> y = set(['a', 'b', 'c', 'd'])
>>> type(y)
<class 'set'>
>>> z = set({'01':'Jan', '02':'Feb', '03':'Mar'})
>>> type(z)
<class 'set'>
```

A Python set is typically created to perform calculations based on the Set Theory, including the mathematical operation of “union”, “intersection”, “difference” between sets. The following are three “set”: `sales`, `tech`, and `managers`. The value “Annie” is a member of sales, tech, and managers sets. “Anita” are members of sales and tech sets. “Jane” is a manager and also a sales. “Mary” is a manager and a tech. “Helen” is a sales, “Lucy” is a tech, and “Willy” is a manager.

```python
>>> sales = set(['Helen', 'Jane', 'Annie', 'Anita'])
>>> tech = set(['Annie', 'Lucy', 'Mary', 'Anita'])
>>> managers = set(['Jane', 'Annie', 'Mary', 'Willy'])
```

The following statement uses the `union` operator (|) to create a new set named “emp” which consists of all elements of sales, tech, and managers. According to the Set Theory, a union set of three individual sets is the set with elements from all sets.

```python
>>> emp = sales | tech | managers
>>> emp
{'Willy', 'Annie', 'Helen', 'Jane', 'Lucy', 'Anita', 'Mary'}
```

The following statement uses the intersection operator (&) to create a new set named “tech_sales” which consists first names that are elements of both sales and tech sets. Only “Anita” and “Annie” are sales and tech.
>>> tech_sales = sales & tech
>>> tech_sales
{'Annie', 'Anita'}

By the same token, the following returns the set of values that appears in both the managers and sales sets.

>>> print(managers & sales)
{'Jane', 'Annie'}

The following uses the difference operator (-) to create a new set named “admin” which consists first names in the “managers” set but not in “sales” and “tech” sets. “Willy” is the only value that appears only in the managers set.

>>> admin = managers - sales - tech
>>> admin
{'Willy'}

The following returns the value that only appear in the sales set.

>>> print(sales - managers - tech)
{'Helen'}

The NoneType

The null keyword is commonly used in many programming languages, such as Java, C++, C# and JavaScript. The None value defined by the “NoneType” class is Python’s equivalent to the null keyword, which is a value that indicates the absence of a value or an object. It is necessary to note that None is an object of the “NoneType” class, such as digits, or True and False, while “NoneType” is a Python data type.

>>> type(None)
<class 'NoneType'>

The following is a statement that assigns the None type to a variable.

>>> s = None
Since None is an object, it cannot check if a variable exists, and it is not an operator to be used to check a condition. In practice, None is used to nullify an object. The term “to nullify” means “to set null value to an object”. In the following example, s is a variable of “str” type which has been assigned a value “Python”. The s variable is then changed to the “NoneType” type to minimize the memory usage. The following example also compares the use of None with the re-assignment of an empty space ("" or ""). The \texttt{sys.getsizeof()} function returns the size of an object in bytes; however, the object must be one of the Python built-in object.

<table>
<thead>
<tr>
<th>Nullification</th>
<th>Assigning a blank space</th>
</tr>
</thead>
</table>
| >>> import sys
>>> s = "Python"
>>> type(s)
<class 'str'>
>>> sys.getsizeof(s)
42
| >>> import sys
>>> t = "Python"
>>> type(t)
<class 'str'>
>>> sys.getsizeof(t)
42 |

<table>
<thead>
<tr>
<th>Nullification</th>
<th>Assigning a blank space</th>
</tr>
</thead>
</table>
| >>> s = None
>>> type(s)
<class 'NoneType'>
>>> sys.getsizeof(s)
8 |
| >>> t = ''
>>> type(t)
<class 'str'>
>>> sys.getsizeof(t)
30 |
In the following, \( x \) is an empty list. By nullifying \( x \), the memory usage is kept to minimum.

```python
>>> import sys
>>> x = ()
>>> sys.getsizeof(x)
28
>>> x = None
>>> sys.getsizeof(x)
8
```

### Finding the data type

The Python `type()` function returns the data type of a given object, while the `id()` function returns a unique identifier of the object assigned by the Python interpreter. The following uses the `type()` function to illustrate how Python implements all objects. The output indicates that 2.1 is a `float` type of value.

```python
>>> type(2.1)
<class 'float'>
```

The `id()` function returns an integer which is the identifier that is guaranteed to be unique and constant for this object during its lifetime.

```python
>>> id(2.1)
20027688
```

The following, for example, returns the integral identifier assigned to the first element of a Python list.

```python
>>> list = [1,2,3]
>>> id(list[0])
1705950792
```

The following samples indicate that date types are `int`, `str`, `bool`, and `NoneType` respectively.

```python
>>> type(47)
<class 'int'>
>>> type('apple')
<class 'str'>
>>> type(True)
<class 'bool'>
>>> type(None)
<class 'NoneType'>
```

Apparently, all values in Python are treated as objects of some data type, with or without an identifier.

### Practical Example

Data types and variables are useful for “problem-solving” parts of programming. For example, the molecular weight of vitamin C (\( \text{C}_6\text{H}_8\text{O}_6 \)) is 176.12g/mol. What is the mass in gram of 0.000142 mole of vitamin C?

In a Chemistry class, the instructor will guide students to solve this problem by multiplying the moles of vitamin C by the molecular weight, as shown below:

\[
\text{mass of vitamin C} = 0.000142 \text{ mole} \times \frac{176.12\text{g}}{1\text{mole}} = 0.025\text{g}
\]

With that being said, the formula used by the Chemistry instructor is:

\[
\text{mass} = \text{molecular weight} \times \text{number of mole}
\]
A Python programmer uses the chemistry instructor’s methodology as a problem-solving algorithm to create a simple application that can solve similar problems. The following is a sample script that declares two variables, \( mw \) and \( nm \), for molecular weight and number of moles respectively, and then perform the calculation to display the result.

\[
\begin{align*}
\text{mw} &= \text{float(input("Enter the molecular weight: "))} \\
\text{nm} &= \text{float(input("Enter the number of mole: "))} \\
\text{print("mass is", mw * nm)}
\end{align*}
\]

As introduced previously, the input() function is a built-in function that takes inputs from keyboard. It is necessary to note that the user’s inputs (taken through the keyboard) are treated as string literal in Python, the float() function is used to convert their data type from string to float.

In Chemistry, molecular weight and number of moles are always expressed as floating-point data (they have fractional part), not integers. Declaring both variables, \( mw \) and \( nm \), as float is more appropriate.

Review Questions

1. Which is not an example of numerical value that is recognized by Python interpreter?
   A. \[3.125e-24\]  
   B. \[0f265\]  
   C. \[6.0e+1234\]  
   D. \[12345678900987654321\]

2. Which is not a legal statement (">>>" is the Python prompt)?
   A. \[>>> x = '6.25e-23'\]  
   B. \[>>> x = "6.25e-23"\]  
   C. \[>>> x = ('6.25e-23')\]  
   D. \[>>> x = print('6.25e-23')\]

3. Which is the expected result of the following statements?
   \[
   \begin{align*}
   \text{>>> PI} &= "3.14159265359" \\
   \text{>>> type(PI)}
   \end{align*}
   \]
   A. <class 'str'>
   B. "3.14159265359"
   C. <class 'float'>
   D. 3.14159265359

4. Which is the expected result of the following statement?
   \[
   \begin{align*}
   \text{>>> type((31/4) > 9)}
   \end{align*}
   \]
   A. False
   B. <class 'bool'>
   C. <class 'float'>
   D. <class 'int'>

5. Given the following statement, Which can return the letter "f"?
   \[
   \begin{align*}
   \text{>>> x} &= "California"
   \end{align*}
   \]
   A. \[x[5]\]  
   B. \[x[len(x)-6]\]  
   C. \[x[len(x)-5]\]  
   D. \[x[6]\]
6. Which can exactly return 'Hello world!' as output?
A. >>> print('Hello', 'world', '!
B. >>> print('Hello world!')
C. >>> 'Hello' + 'world' + '!
D. >>> 'Hello' + " " + 'world' + '!

7. Given the following statement, Which is the expected result?
>>> s = "Python Programming"
>>> s[7:10]
A. 'Pro'
B. 'Pr'
C. 'rog'
D. 'nP'

8. Given the following statements, Which is the expected result?
>>> x = 5
>>> x < 4
False
>>> type(x)
A. <class 'bool'>
B. <class 'str'>
C. <class 'int'>
D. False

9. Given the following statements, which object is mutable?
>>> w = 'apple'
>>> x = ['orange']
>>> y = ('banana')
>>> z = "cherry"
A. w
B. x
C. y
D. z

10. Which has the lowest memory usage?
A. x = 
B. x = ""
C. x = 'None'
D. x = None
Python

Lab #2     Python Data Type and Variables

Learning Activity #1
1. Open the Command Prompt (or Terminal Emulator for Linux user).

<table>
<thead>
<tr>
<th>Microsoft OS</th>
<th>Linux</th>
</tr>
</thead>
<tbody>
<tr>
<td>2. In the prompt, type <code>cd c: \cis247</code> and press [Enter] to change to the &quot;cis247&quot; directory.</td>
<td>2. In the shell prompt, type <code>cd ~/cis247</code> and press [Enter] to change the “cist247” directory.</td>
</tr>
<tr>
<td>C: \Users\user&gt; cd c: \cis247</td>
<td>$ cd ~/cis247</td>
</tr>
<tr>
<td>3. Under the “cis247” directory, type python (or python3 for Linux user) and press [Enter] to launch the Python interpreter. “&gt;&gt;&gt;” is the Python prompt.</td>
<td>3. Type <code>python3</code> and press [Enter].</td>
</tr>
<tr>
<td>C: \cis247&gt; python Python 3.7.2 (v3.7.2:37dagde53f8s9, Sep 1 2018, 02:16:59) [MSC v.1900 32 bit (Intel)] on win32 Type &quot;help&quot;, &quot;copyright&quot;, &quot;credits&quot; or &quot;license&quot; for more information. &gt;&gt;&gt;</td>
<td>$ python3 ......</td>
</tr>
</tbody>
</table>

4. In the Python prompt (“>>>”), try the following statements, one at a time, to observe how Python interpreter display literals.

```python
>>> "16.9"
16.9
>>> '37'
37
>>> "Python"
'Python'
>>> 'programming'
'programming'
```

5. Try the following statements, one at a time, to understand what the 'int' type of data are.

```python
>>> 17
17
>>> 44.5
44.5
>>> 6.25e-23
6.25e-23
```

6. Try the following statements, one at a time, to use type() and id() functions to check the data type of objects.

```python
>>> type(2.1)
<class 'float'>
>>> id(2.1)
20027688
>>> type(47)
<class 'int'>
>>> type('apple')
<class 'str'>
>>> type(True)
<class 'bool'>
>>> type(None)
<class 'NoneType'>
```
7. Try the following statements, one at a time, to understand how to: (a) designate an identifier to a variable; (b) assign a value to the variable; and (c) set the data type of the variable.

```python
>>> x = 2.1
>>> type(x)
<class 'float'>
>>> id(x)
20027688
>>> y = "orange"
>>> type(y)
<class 'str'>
```

8. Type `exit()` and press [Enter] to exit Python prompt.

```python
>>> exit()
E\cis247>
```

9. Under the “cis247” directory, type `notepad lab2_1.py` (or “gedit lab2_1.py” for Linux user) and press [Enter] to create a new file.

```bash
c:\cis247>notepad lab2_1.py
```

10. In the Notepad (or “gedit”) window, type the following Python code.

```python
s = "Result:\n"
tuple1 = ("16.9", '37', "Python", 'programming')
s += str(tuple1) + "\n" # example 1

tuple2 = (17, 44.5, 6.25e-23)
s += str(tuple2) + "\n" # example 2

#example 3
tuple3 = (type(2.1), id(2.1), type(47), type('apple'), type(True), type(None))
s += str(tuple3) + "\n"

#example 4
x = 2.1
s += str(type(x)) + "\n"
s += str(id(x)) + "\n"
y = "orange"
s += str(type(y)) + "\n"

##### Message Box Code ######
from tkinter import *

root = Tk()
root.title('Message Box')
Label(root, justify=LEFT, text = s).grid(padx=10, pady=10)

root.mainloop()
```

11. Type `python lab2_1.py` and press [Enter] to test the program. A sample output looks:
12. Download the “assignment template”, and rename it to lab2.doc if necessary. Capture a screen shot similar to the above and paste it to a Microsoft Word document named “lab2.doc” (or .docx).

Learning Activity #2
1. In the Python prompt, try the following statements, one at a time, to set a variable to be “int” type first, then “str” and “tuple” accordingly.

```python
>>> x = 2
>>> type(2)
<class 'int'>
>>> x = "Python"
>>> type(x)
<class 'str'>
>>> x = (9, 8, 7, 6)
>>> type(x)
<class 'tuple'>
```

2. Try the following statements, one at a time, to use one variable name to represent different kinds of objects at different times.

```python
>>> x = 0            # x bound to an integer object
>>> x = "Hello"      # Now it's a string.
>>> x = [1, 2, 3]    # And now it's a list.
```

3. Try the following statements, one at a time, to use scientific notation and numbers of non-decimal formats.

```python
>>> 3.14e-10
3.14e-10
>>> 5.07E210
5.07E210
>>> 0b1111100  # binary
124
>>> 0o174   # octal
124
>>> 0x7c   # dexadecimal
124
>>> 1, 10, 100  # decimal literals
(1, 10, 100)
>>> 0b1 , 0b10, 0b100  # binary literals
(1, 2, 4)
>>> 0o1, 0o10, 0o100  # Octal literals
(1, 8, 64)
>>> 0x01, 0x10, 0xFF  # Hex literals
(1, 16, 255)
```

4. Try the following statements, one at a time, to observe the case in-sentiveness of digits and exponent sign.

```python
>>> 0XFF == 0Xff == 0XFF == 0xFF
```
True

```python
>>> 0b1011, 0B1011
(11, 11)
>>> 0o177, 0O177
(127, 127)
>>> 0x9ff, 0X9ff
(2559, 2559)
>>> 5.07e+210 == 5.07e210
True
>>> 5.07e+210 == 5.07E210
True
```

5. Type `exit()` and press [Enter] to exit Python prompt.

6. Under the “cis247” directory, use Notepad (or “gedit” for Linux user) to create a new file named `lab2_2.py` with the following content.

```python
s = "Result:
"
x = 2   # example 1
s += str(type(2)) + "\n"
x = "Python"
s += str(type(x)) + "\n"
x = (9, 8, 7, 6)
s += str(type(x)) + "\n"
x = 0  # example 2
s += str(x) + "\n"
x = "Hello"
s += str(x) + "\n"
x = [1, 2, 3]
s += str(x) + "\n"
tuple1 = (3.14e-10, 5.07E210, 5.07e+210)
s += str(tuple1) + "\n" # example 3
tuple2 = (0b1111100, 0o174, 0x7c)
s += str(tuple2) + "\n"
tuple3 = (1, 10, 100)
s += str(tuple3) + "\n"
tuple4 = (0b1 , 0b10, 0b100)
s += str(tuple4) + "\n"
tuple5 = (0o1, 0o10, 0o100)
s += str(tuple5) + "\n"
tuple6 = (0x01, 0x10, 0xFF)
s += str(tuple6) + "\n"
s += str(0XFF == 0Xff == 0XFf == 0XfF) + "\n" #example 4
tuple7 = (0b1011, 0B1011)
s += str(tuple7) + "\n"
tuple8 = (0o177, 0O177)
s += str(tuple8) + "\n"
```
tuple9 = (0x9ff, 0X9ff)
s += str(tuple9) + "\n"

s += str(5.07e+210 == 5.07e210) + "\n"
s += str(5.07e+210 == 5.07E210) + "\n"

######## Message Box Code ########
from tkinter import *

root = Tk()
root.title('Message Box')
Label(root, justify=LEFT, text = s).grid(padx=10, pady=10)
root.mainloop()

7. Test the program. A sample output looks:

![Message Box Output](image)

8. Capture a screen shot similar to the above and paste it to a Microsoft Word document named “lab2.doc” (or .docx).

Learning Activity #3:
1. In the Python prompt, try the following statements, one at a time, to use the oct() function converts decimal to octal, and the hex() to hexadecimal as well as the int() and eval() functions.

   >>> oct(64), hex(64), hex(255)
   ('0o100', '0x40', '0xff')

   >>> int('0100'), int('0100', 8), int('0x40', 16)
   (100, 64, 64)

   >>> eval('100'), eval('0o100'), eval('0x40')
   (100, 64, 64)

2. Try the following statements, one at a time, to observe how Python handles complex numbers.

   >>> 1j * 1j
   (-1+0j)

   >>> 2 + 1j * 3
   (2+3j)

   >>> (2+1j)*3
   (6+3j)

   >>> (3 + 5j) * (2 - 1j)
(11 + 7j)

3. Try the following statements, one at a time, to observe how different types of quotes handle Python string literals.

```python
>>> 'S1562'  # single quotes
'S1562'
>>> "Classic Jazz music"    # double quotes
'Classic Jazz music'
>>> '''Recyle the trash to save the earth'''
'Recyle the trash to save the earth'
>>> """Taipei 101""
'Taipei 101'
>>> print('''Content-type: text/html
... <h1> Hello World </h1>
... Click <a href="http://www.python.org">here</a>.
... ''')
Content-type: text/html
<h1> Hello World </h1>
Click <a href="http://www.python.org">here</a>
>>> "Python" + " 3.4" + "programming"
'Python programming'
>>> print("Python", "3.4", "programming")
Python 3.4 programming
```

4. Try the following statements, one at a time, to observe how a string literal is in essence a sequence type of data.

```python
>>> x = "happy hours"
>>> len(x)
11
>>> x[0], x[1], x[2], x[3], x[4], x[5], x[6], x[7], x[8], x[9], x[10]
('h', 'a', 'p', 'p', 'y', ' ', 'h', 'o', 'u', 'r', 's')
>>> x[4]
y
>>> x[len(x)-3]
u
>>> x[0:5]
'happy'
>>> x[2:5]
'ppy'
>>> x[5:]
'happy'
>>> x[6:]
'hours'
```

5. Type `exit()` and press [Enter] to exit Python prompt.

6. Under the “cis247” directory, use Notepad (or “gedit” for Linux user) to create a new file named `lab2_3.py` with the following content.

```python
s = "Result:\n"
tuple1 = (oct(64), hex(64), hex(255)) #example 1
s += str(tuple1) + "\n"
tuple2 = (int('0100'), int('0100', 8), int('0x40', 16))
s += str(tuple2) + "\n"
tuple3 = (eval('100'), eval('0o100'), eval('0x40'))
s += str(tuple3) + "\n"
```
tuple4 = (1j * 1J, 2 + 1j * 3, (2+1j)*3, (3 + 5j) * (2 - 1j))  # example 2
s += str(tuple4) + "\n"

s += str('S1562') + "\n"  # example 3
s += str("Classic Jazz music") + "\n"

s += str('''Recyle the trash to save the earth'''') + "\n"

s += str("""Taipei 101"""") + "\n"

s += str('''Content-type: text/html
<h1> Hello World </h1>
  Click <a href="http://www.python.org">here</a>.</r
''') + "\n"

s += str("Python" + " " + "programming") + "\n"

s += str(("Python", "3.4", "programming")) + "\n"

# example 4
x = "happy hours"
s += str(len(x)) + "\n"
tuple5 = (x[0], x[1], x[2], x[3], x[4], x[5], x[6], x[7], x[8], x[9], x[10])
s += str(tuple5) + "\n"

s += str(x[4]) + "\n"

s += str(x[len(x)-3]) + "\n"

s += str(x[0:5]) + "\n"

s += str(x[2:5]) + "\n"

s += str(x[:5]) + "\n"

s += str(x[6:]) + "\n"

# Message Box Code
from tkinter import *

root = Tk()
root.title('Message Box')
Label(root, justify=LEFT, text = s).grid(padx=10, pady=10)
root.mainloop()

7. Test the program. A sample output looks:
Learning Activity #4

1. In the Python prompt, try the following statements, one at a time, to use the str() or repr() function for type conversion.

   ```python
   >>> y = 63.5
   >>> type(y)
   <class 'float'>
   >>> type(str(y))
   <class 'str'>
   >>> type(str(3.14))
   <class 'str'>
   >>> type(repr(7.12))
   <class 'str'>
   >>> 'Python programming ' + str(3) + " units"
   'Python programming 3 units'
   >>> 'Python programming ' + repr(3) + " units"
   'Python programming 3 units'
   ```

2. Try the following statements, one at a time, to uses upper() and lower() functions to change cases.

   ```python
   >>> str = "Welcome To Python!"
   >>> print(str.upper())
   WELCOME TO PYTHON!
   >>> print(str.lower())
   welcome to python!
   ```

3. Try the following statements, one at a time, to uses upper() and lower() functions to change cases.

   ```python
   >>> str = "Welcome To Python!"
   >>> print(str.upper())
   WELCOME TO PYTHON!
   >>> print(str.lower())
   welcome to python!
   ```

4. Try the following statements, one at a time, to understand what Boolean values are.

   ```python
   >>> type(True)
   <class 'bool'>
   >>> type(False)
   <class 'bool'>
   >>> 3 > 2
   True
   ```
>>> x = 5
>>> x < 4
False
>>> type(x)
<class 'int'>
>>> x = 5
>>> y = (x < 4)
>>> type(y)
<class 'bool'>
>>> s = "D901"
>>> s.isalpha()
False

5. Type exit() and press [Enter] to exit Python prompt.

6. Under the “cis247” directory, use Notepad (or “gedit” for Linux user) to create a new file named lab2_4.py with the following content.

```python
s = "Result: \n"
y = 63.5   #example 1
s += str(type(y)) + "\n"
s += str(type(str(y))) + "\n"
s += str(type(str(3.14))) + "\n"
s += str(type(repr(7.12))) + "\n"
s += str('Python programming ' + str(3) + " units") + "\n"
s += str('Python programming ' + repr(3) + " units") + "\n"
str1 = "Welcome To Python!"   #example 2
s += str(str1.upper()) + "\n"
s += str(str1.lower()) + "\n"
str1 = "Welcome To Python!"    #example 3
s += str(str1.upper()) + "\n"
s += str(str1.lower()) + "\n"
s += str(type(True)) + "\n"  #example 4
s += str(type(False)) + "\n"
s += str(3 > 2) + "\n"
x = 5
s += str(x < 4) + "\n"
s += str(type(x)) + "\n"
x = 5
y = (x < 4)
s += str(type(y)) + "\n"
w = "D901"
s += str(w.isalpha()) + "\n"

#### Message Box Code ####
from tkinter import *
```
root = Tk()
root.title('Message Box')
Label(root, justify=LEFT, text = s).grid(padx=10, pady=10)
root.mainloop()

7. Test the program. A sample output looks:

```
Results:
<class 'float'>
<class 'str'>
<class 'str'>
Python programming 3 units
Python programming 3 units
WELCOME TO PYTHON!
welcome to python!
WELCOME TO PYTHON!
welcome to python!
<class 'bool'>
<class 'str'>
True
False
<class 'int'>
<class 'bool'>
False
```

8. Capture a screen shot similar to the above and paste it to a Microsoft Word document named “lab2.doc” (or .docx).

Learning Activity #5:
1. Make sure both “MessageBox.py” and “InputBox.py” files are in the “cis247” directory. (See Appendix of Lab #1 for details).

2. Under the “cis247” directory, use Notepad (or “gedit” for Linux user) to create a new file named lab2_5.py with the following content.

```python
import InputBox
InputBox.ShowDialog("What is your full name?")
fullname = InputBox.GetInput()

InputBox.ShowDialog("What is your major?")
major = InputBox.GetInput()

InputBox.ShowDialog("What is the current temperature in Fahrenheit?")
f = InputBox.GetInput()
c = (float(f) - 32) * (5/9)
s = "Results:
#example 1 - list
l1 = [3, 6, 9]
s += str(l1) + "\n"

l2 = ['A', 'B', 'C', 'D']
s += str(l2) + "\n"

l3 = ['Python', 3.4, True]
s += str(l3) + "\n"
#example 2 - tuple
```
t1 = (3, 6, 9)
s += str(t1) + "\n"

t2 = ('A', 'B', 'C', 'D')
s += str(t2) + "\n"

t3 = ('Python', 3.4, True)
s += str(t3) + "\n"

#example 3 - dictionary
d={'OH':"Ohio", 'TX':"Texas"}
s += str(d) + "\n"

#example 4
ans = [] #empty list
ans.append(fullname)
ans.append(major)
ans.append(f)
s += str(ans) + "\n"
s += "It is " + str(c) + "degrees in Celsius.\n"

import MessageBox
MessageBox.Show(s)

3. Test the program. A sample output looks:

4. Capture a screen shot similar to the above and paste it to a Microsoft Word document named “lab2.doc” (or .docx). [Feel free to convert the Word document to a .pdf file.]

Submittal
1. Upon the completion of learning activities, compress the following files and name the zipped file lab2.zip:
   - Lab2_1.py
   - Lab2_2.py
   - Lab2_3.py
   - Lab2_4.py
   - Lab2_5.py
   - Lab2.doc (or .docx or .pdf)

2. Upload the zipped file as response to Question 11 (available at Blackboard).

Python Programming – Penniel P. Wu, PhD. 70
Programming Exercise #02:
1. Create a Python source file named “ex02.py” that will include the following two lines. Be sure to replace [YourNameHere] with your full name.

```
# Student: YourNameHere
# Programming Exercise: 02
```

2. Next to the above two lines, write Python codes that imports both “InputBox.py” and “MessageBox.py”. Then display a dialog box to ask the user to enter how much pounds she/he weights in pounds, convert the value to kilograms, and then display the result as a string similar to “You weight xxx.xxxx kg.” in a message box, as shown below. [Hint: 1 pound is equal to 0.45359237 kilogram.]

```
Input Box
How much do you weigh in pounds?
153.25
OK
```

```
MessageBox
M  
You weigh 88.5684461662 kg.
```

Fig. ex02

3. Download the “programming exercise template”, and rename it to ex02.doc. Copy and paste the source code to the template. Capture a screen shot similar to Fig. ex02 and paste it to a Microsoft Word document named ex02.doc (or ex02.docx). [Feel free to convert the Word document to a .pdf file.]

4. Compress both .py and .doc (or .docx or .pdf) to a .zip file named ex02.zip.

5. Upload the ex02.py file as response of Question 12.

Note: Your code must be fully functional to earn the credit. No partial credit is given. You will receive zero points if either .py or .doc (or .docx) file is not submitted.