Introduction to Python Programming

Lec-5

Variables and Assignment Statements

- As in algebra, Python expressions also may contain variables, which store values for later use in your code. Let's create a variable named x that stores the integer 7, which is the variable's value:

```
In [2]: x = 7
```

```
In [3]: y = 3
```

- Snippet [2] is a statement. Each statement specifies a task to perform. The preceding statement creates x and uses the assignment symbol (=) to give x a value. The entire statement is an assignment statement that we read as "x is assigned the value 7." Most statements stop at the end of the line, though it's possible for statements to span more than one line.

Adding Variable Values and Viewing the Result

```
In [4]: x + y
Out[4]: 10
```

- The + symbol is the addition operator. It’s a binary operator because it has two operands (in this case, the variables x and y) on which it performs its operation.

Calculations in Assignment Statements

- You'll often save calculation results for later use. The following assignment statement adds the values of variables x and y and assigns the result to the variable total, which we then display

```
In [5]: total = x + y

In [6]: total
Out[6]: 10
```

Variable Naming and Types

- A variable name, such as x, is an identifier. Each identifier may consist of letters, digits and underscores (_) but may not begin with a digit. Python is case sensitive, so number and Number are different identifiers because one begins with a lowercase letter and the other begins with an uppercase letter.
- Types Each value in Python has a type that indicates the kind of data the value represents. You can view a value's type, as in:

```
In [7]: type(x)
Out[7]: int

In [8]: type(10.5)
Out[8]: float
```

Arithmetic

- These are the arithmetic operators used in Python.
- +, -, *, /, //, **, %
- Python supports / and //. Dividing 2 ints with a / will return a float.
- To return an int use //. (Dividing 2 floats or an int with a float and vice versa with // will return a float)

Exceptions and Tracebacks

- Exceptions and Tracebacks Dividing by zero with / or // is not allowed and results in an exception—a sign that a problem occurred:

```
In [10]: 123 / 0
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-10-cd759d3fcf39> in <module>()
----> 1 123 / 0

ZeroDivisionError: division by zero
```

- Python reports an exception with a traceback. This traceback indicates that an exception of type ZeroDivisionError occurred—most exception names end with Error.

List of Errors *

- ZeroDivisionError: division by zero
- NameError: name ‘_’ is not defined
- SyntaxError: invalid syntax
- ValueError: invalid literal for int() with base 10: 'hello’

Grouping Expressions with Parentheses

- Parentheses group Python expressions, as they do in algebraic expressions.

```
In [14]: 10 * (5 + 3)
Out[14]: 80
```

```
In [15]: 10 * 5 + 3
Out[15]: 53
```

- The parentheses are redundant (unnecessary) if removing them yields the same result.

Operator Precedence and Associativity

- Python applies the operators in arithmetic expressions according to the following rules of operator precedence. These are generally the same as those in algebra.
- Expressions in parentheses evaluate first, so parentheses may force the order of evaluation to occur in any sequence you desire. Parentheses have the highest level of precedence. In expressions with nested parentheses, such as (a / (b- c)), the expression in the innermost parentheses (that is, b-c) evaluates first.
- Exponentiation operations evaluate next. If an expression contains several exponentiation operations, Python applies them from right to left.
- Multiplication, division and modulus operations evaluate next. If an expression contains several multiplication, true-division, floor-division and modulus operations, Python applies them from left to right. Multiplication, division and modulus are “on the same level of precedence.”
- Addition and subtraction operations evaluate last. If an expression contains several addition and subtraction operations, Python applies them from left to right. Addition and subtraction also have the same level of precedence.

Function print and an Intro to Single- and Double Quoted Strings

- The built-in print function displays its argument(s) as a line of text.

```
In [1]: print('Welcome to Python!')
Welcome to Python!
```

- In this case, the argument 'Welcome to Python!' is a string—a sequence of characters enclosed in single quotes ('). Unlike when you evaluate expressions in interactive mode, the text that print displays here is not preceded by Out[1].

Printing a Comma-Separated List of Items

- The print function can receive a comma-separated list of arguments, as in:

```
In [3]: print('Welcome', 'to', 'Python!')
Welcome to Python!
```

Printing Many Lines of Text with One Statement

- Printing Many Lines of Text with One Statement When a backslash (\) appears in a string, it's known as the escape character. The backslash and the character immediately following it form an escape sequence. For example, \n rep resents the newline character escape sequence, which tells print to move the output cursor to the next line. Placing two newline characters back-to-back displays a blank line.

```
In [4]: print('Welcome\nto\n\nPython!')
Welcome
to

Python!
```

Most widely used escape sequences are
\n \t
\\
\' \"

Ignoring a Line Break in a Long String and Performing Calculations inside print()

- You may also split a long string (or a long statement) over several lines by using the '\' continuation character as the last character on a line to ignore the line break:

```
In [5]: print('this is a longer string, so we \
   ...: split it over two lines')
this is a longer string, so we split it over two lines
```

- Calculations can be performed in print statements:

```
In [6]: print('Sum is', 7 + 3)
Sum is 10
```

Triple Quoted Strings

- Triple-quoted strings begin and end with three double quotes (""") or three single quotes ('‘’).
- Use these to create:

  multiline strings,

  strings containing single or double quotes and

  docstrings, which are the recommended way to document the purposes of certain program components.
- In a string delimited by single quotes, you may include double-quote characters

```
In [1]: print('Display "hi" in quotes')
Display "hi" in quotes
```

Triple Quoted Strings

- but not single quotes

```
In [2]: print('Display 'hi' in quotes')
  File "<ipython-input-2-19bf596ccf72>", line 1
    print('Display 'hi' in quotes')
                     ^
SyntaxError: invalid syntax
```

- unless you use the \' escape sequence

```
In [3]: print('Display \'hi\' in quotes')
Display 'hi' in quotes
```

Multiline Strings

- The following snippet assigns a multiline triple-quoted string to the variable triple_quoted_string:

```
In [7]: triple_quoted_string = """This is a triple-quoted
   ...: string that spans two lines"""
```

- IPython knows that the string is incomplete because we did not type the closing """ before we pressed Enter. So, IPython displays a continuation prompt ...: at which you can input the multiline string's next line. This continues until you enter the ending """ and press Enter.