PennState World Campus

DAAN862: Analytics Programming in Python

Lesson 1: Introduction to Python

Lesson 1: Objectives and Overview (1 of 7)

Lesson 1: Introduction to Python

This lesson is a quick introduction to the Python programming language and use either Spyder or Jupyter Notebook.

At the end of this lesson, students will be able to:

- Download and install Python on the computer
- Define and utilize Python basic structure for basic data types, expressions and operators, and control flow statements

By the end of this lesson, please complete all readings and assignments found in the Lesson 1 Course Schedule.
Check the **Course Syllabus** to access the book chapters online from O'reilly.

Lesson 1.1: Installation of Python (2 of 7)

Lesson 1.1: Installation of Python

Why Python for Data Analysis?

- Python is one of the most popular programming languages in both academia and industry settings
- A quick scripting language for small programs and heavy tasks
- Powerful data mining and matching learning libraries such as pandas and scikit-learn

Installation

Python is an interpreted language which requires an interpreter to execute a program by running one statement each time. In this course, we will use **Spyder** as the interpreter for Python.

- **Step 1:** Download Python 3.7 version or later from this website (https://www.anaconda.com/download/) .

Show Image

Fig. 1.1

- **Step 2:** You could create a shortcut folder on your desktop (i.e.DAAN862). You can click either 64-Bit or 32-Bit Graphical Installer based on your computer, and save the download into the created folder (i.e.DAAN862).

  Show Image

Fig. 1.2

- **Step 3:** Install Anaconda. The installation of Anaconda is very easy. You can simply go by default after you click Anaconda 3-5.1.0-Windows-x86_64.exe installer file. Fig. 1.3 highlights the installation of Python process.

  Show Image

Fig. 1.3

- **Step 4:** Start → Anaconda3 → Spyder to open Python console. Figure 1.4 shows how Spyder looks.

  Show Image

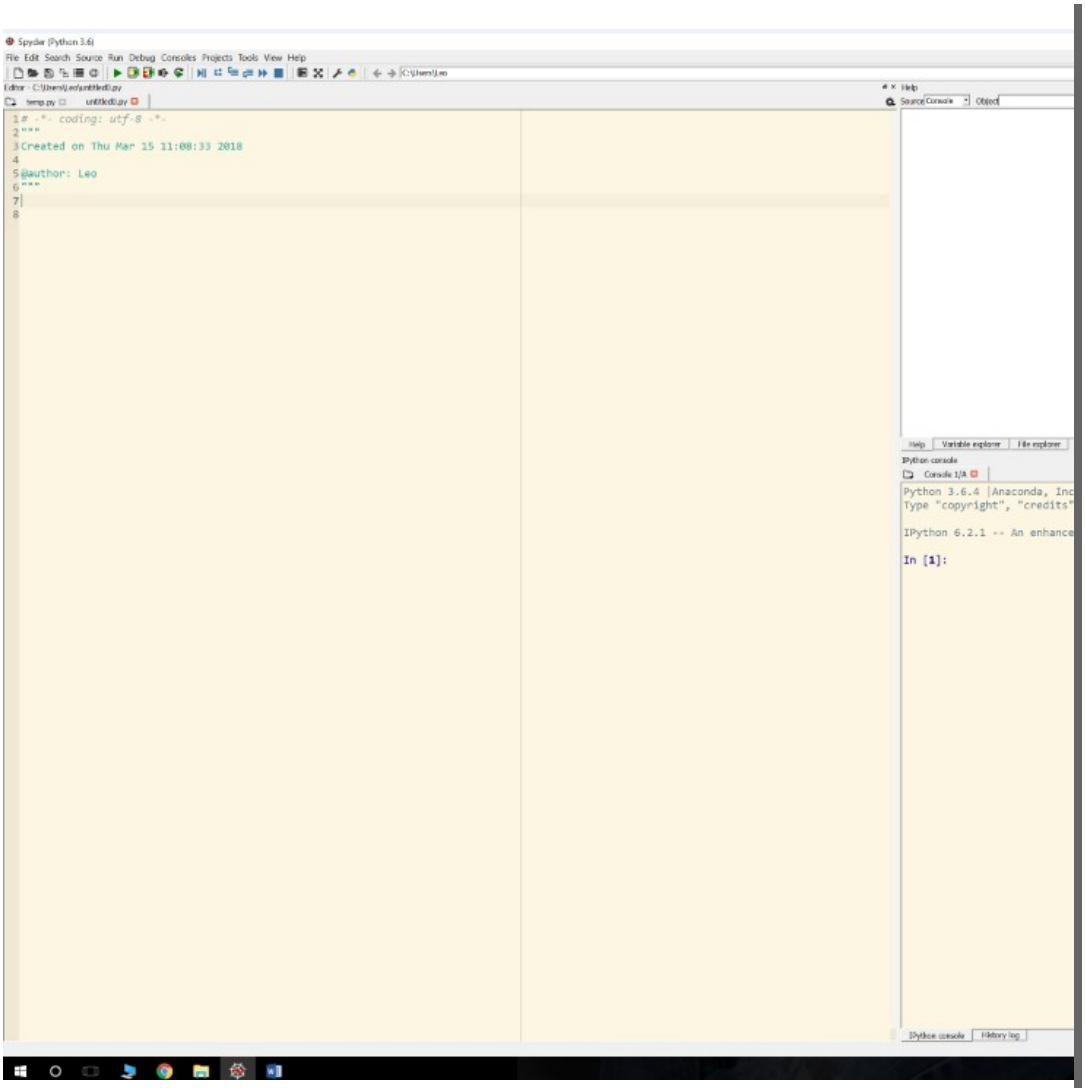

Fig. 1.4 (click to enlarge image

Lesson 1.2: Python Language Basics

This section will cover an overview of basic Python programming concepts, language mechanics, and basic data types.

(click the titles to learn more)

Lines and Indentation

A Python program consists of one or more lines. Everything after the # and to the end of the line will be considered as comments.

Although, some programming languages utilize braces or delimeters to express the blocks of a program, python uses indentation, which is the only way to separate blocks.

Below you can find an example of a **Block**:

A colon means the start of the block (content within the for loop)

All codes inside the block have the same amount of indentation

```
for x in array:
    if x < 0:
        print("Negative")
    elif x > 0:
        print("Positive")
```

Fig 1.5 (click to enlarge)

In Python, you can use the tab button on your keyboard to create an indentation which will typically contain four white spaces.

Variables and Argument Passing

In Python, the equal sign (=) is used to assign values to a variable. For example, you can assign 0 to the variable *a* as seen in Figure 1.6:

```
In [1]: a = 0                    # assign 0 to variable a
```

Fig 1.6 (click to enlarge)

Figure 1.7 shows how you can assign the same value to multiple variables by using *a = b = c = 0*:

```
In [2]: a = b = c = 0            # assign 0 to variable a, b, c at same time
```

Fig 1.7 (click to enlarge)

You can also assign different values to multiple values by order as seen in Figure 1.8:

```
In [3]: a = 1; b = 2; c = 2      # Use semicolon to separate multiple statement

In [4]: a, b, c = 1, 2, 3        # equavelent to previous line
```

Fig 1.8 (click to enlarge)

Python provides a very convenient way to swap values of two variables as seen in Figure 1.9:

```
In [5]: a, b = b, a              # swap the value for a and b

In [6]: print(a, b)
2 1
```

Fig 1.9 (click to enlarge)

Scalar Types

Scalar data types are referred to as single-value. Python has several built-in scalar types such as numeric values, strings, Booleans, etc. See Table 1.2.1 for the list of standard Python Scalar Types:

Table 1.2.1

Type	Description
None	The Python “null” value (only one instance of the None object exists)
str	String type; holds Unicode (UTF-8 encoded) strings
bytes	Raw ASCII bytes (or Unicode encoded as bytes)
float	Double-precision (64-bit) floating-point number
bool	A True or False value
int	Arbitrary precision signed integer

In this section, we only introduce numeric types, strings, boolean, and None.

(click the tabs to learn more)

Numeric Types:

The main Python types for numbers are **int** and **float**.

Int:

int is the data type for integers

```
In [7]: int1 = 123456789

In [8]: int1**5
Out[8]: 2867971860299718107233761438093672048294 9
In [9]: type(int1)
Out[9]: int
```

Fig 1.10 (click to enlarge)

Float:

float is the data type for decimal numbers

```
In [10]: float1 = 3.1415

In [11]: float2 = 5.69e-6          # use scitific notation

In [12]: type(float1)
Out[12]: float

In [13]: type(float2)
Out[13]: float
```

Fig 1.11 (click to enlarge)

String:

A ***String*** is a sequence of characters used to represent text. A string can be single quoted or double-quoted, but you should not mix them. This is a similar data type as the character in R programming language.

Figure 1.12 illustrates the examples to create string variables:

```
In [14]: s1 = 'Hello, world'           # Single quotations

In [15]: s2 = "Hello, world"           # Double quotations

In [16]: type(s1)
Out[16]: str

In [17]: s1 == s2
Out[17]: True
```

Fig 1.12 (click to enlarge)

You can use triple quotes, either "" or """ , for multiple strings with line breaks as seen in Figure 1.13:

```
In [18]: c = """This is a longer string that
    ...: spans multiple lines"""

In [19]: c
Out[19]: 'This is a longer string that \nspans multiple lines'
```

Fig 1.13 (click to enlarge)

Figure 1.14 shows how you can add two strings together to produce a new string:

```
In [20]: a = "The first string "

In [21]: b = "and the second string"

In [22]: a + b
Out[22]: 'The first string and the second string'
```

Fig 1.14 (click to enlarge)

Booleans:

In Python, the two ***Boolean*** values are written as *True* and *False*. Boolean values can be used for logic operations such as "and" and "or":

```
In [23]: x = True

In [24]: y = False

In [25]: x and y
Out[25]: False

In [26]: x or y
Out[26]: True
```

Fig 1.15 (click to enlarge)

None:

None is the Python null value type. Please be aware that if a function doesn't return anything by default it will return *None*:

```
In [21]: x = None

In [22]: y = 3

In [23]: x is None
Out[23]: True

In [24]: y is not None
Out[24]: True
```

Fig 1.16 (click to enlarge)

You can use type function to check the data type (shown in Figure 1.17):

```
In [25]: type(x)
Out[25]: NoneType

In [26]: type(y)
Out[26]: int
```

Fig 1.17 (click to enlarge)

Expression and Operators

Please see Table 1.2.2 for common expressions and operations:

Table 1.2.2

Operator	Description
'exp,…'	String conversion
{key : exp, …}	Dictionary creation
[exp, …]	List creation
(expr, …)	Tuple creation or just parentheses
f(expr, …)	Function call
x[index : index]	Slicing
x[index]	Indexing
x.attr	Attribute reference
xy**	Exponentiation (x to yth power)
~x	Bitwise Not

Operator	Description
+x, -x	Unary plus and minus
x*y, x/y, x//y, x%y	Multiplication, division, truncating division, remainder
x+y, x-y	Addition, subtraction
x<<y, x>>y	Left shift, right-shift
x&y	Bitwise AND
x^y	Bitwise XOR
x|y	Bitwise OR
x<y, x>y, x<=y, x>= y, x==y, x!=y	Comparison
x is y, x is not y	Identify tests
x in y, x not in y	Membership tests
not x	Boolean Not
x and y	Boolean AND
x or y	Boolean OR
lambda arg, …, : expr	Anonymous simple function

Print Statement

Print function can be used to print one or more expressions which are separated by commas. This is a very convenient way to show results in the text format.

Figure 1.18 shows examples of print statements:

```
In [31]: letter = 'c'

In [32]: print(letter )
c

In [33]: number = 42

In [34]: print("the number is:", number)             # prints "the number is: 42"
the number is: 42
```

Fig 1.18 (click to enlarge)

Lesson 1.3: Basic Data Type and Built-in Methods (4 of 7)

Lesson 1.3: Basic Data Type and Built-in Methods

Python contains very powerful built-in data structures such as *lists*, *tuples*, *sets*, and *dicts*.

(click the titles to learn more)

List

In Python, a list is a series of items separated by commas within brackets ([ ]). Lists are mutable which means you can modify them. The elements in lists can have different data types or data structures.

List Creation:

Figure 1.19 shows some examples of how to create lists:

```
In [1]: [1, 2, 3]                  # List with three items with the same type
Out[1]: [1, 2, 3]

In [2]: [4, 5.67, "Python"]        # List with three items but they are different
types
Out[2]: [4, 5.67, 'Python']

In [3]: [45]                       # List with one item
Out[3]: [45]

In [4]: []                         # Empty list
Out[4]: []
```

Figure 1.19 (click to enlarge)

Indexing and Slicing:

Python uses 0-based index, which means the index for the first element of a list is 0. You can index a list by using a format like *Listname* [index] (as shown in Figure 1.20):

```
In [5]: L = [12, 90, 4, 5.67, "Python"]

In [6]: L[0]                        # Select the first element
Out[6]: 12

In [7]: L[2]                        # Select the third element
Out[7]: 4
```

Fig 1.20 (click to enlarge)

You can slice the list by using *ListName* [start:stop] (shown in Figure 1.21):

```
In [8]: L[1:4]                     # Select from the second to the fourth elements
Out[8]: [90, 4, 5.67]

In [9]: L[:3]                      # Select from the first element to the third
element
Out[9]: [12, 90, 4]

In [10]: L[2:]                      # Select from the third element to the end
Out[10]: [4, 5.67, 'Python']
```

Fig 1.21 (click to enlarge)

It also provides a simple way to reverse a list (shown in Figure 1.22):

```
In [11]: L[::-1]                    # reverse the list
Out[11]: ['Python', 5.67, 4, 90, 12]
```

Fig 1.22 (click to enlarge)

List Methods:

Please refer to Table 1.3.1 for all list methods.

Table 1.3.1

Method	Description
L.append(x)	Add an item to the end of the list
L.extend(iterable)	Extend the list by appending all items from the iterable
L.insert(i,x)	Insert an item at a given position
L.remove(x)	Remove the first item from the list whose value is x. It is an error if there is no such item
L.pop([i])	Remove the item at the given position in th list, and return it. If no index is specified, a.pop() removes and returns the last itme in the list
L.index(x)	Return zero-based index in the list of the first item whose value is x. Raises a ValueError if there is no such item
L.count(x)	Return the number of times x appears in the list
L.reverse()	Reverse the elements of the list in place
L.sort (key=None,reverse=False)	Sort the items of the list in place (the arguments can be used for sort customization, see sorted () for their explanation)

Below you can find examples of how to apply the methods listed in Table 1.3.1

```
In [13]: L = [3, 7, 2, 56, 23, 21]
    ...: L.append(21)               # Append 21 at the end of L

In [14]: L
Out[14]: [3, 7, 2, 56, 23, 21, 21]

In [15]: L.extend([3, 8])           # Append 3 and 8 at the end of L

In [16]: L
Out[16]: [3, 7, 2, 56, 23, 21, 21, 3, 8]

In [17]: L.insert(4, 100)           # Insert 100 as the fifth element

In [18]: L
Out[18]: [3, 7, 2, 56, 100, 23, 21, 21, 3, 8]

In [19]: L.remove(56)               # Remove 56 from L

In [20]: L
Out[20]: [3, 7, 2, 100, 23, 21, 21, 3, 8]

In [21]: L.pop()                    # Remove the first element
Out[21]: 8

In [22]: L
Out[22]: [3, 7, 2, 100, 23, 21, 21, 3]

In [23]: L.pop(3)                   # Remove the fourth element
Out[23]: 100

In [24]: L
Out[24]: [3, 7, 2, 23, 21, 21, 3]

In [25]: L.index(23)                # Return the index of 23
Out[25]: 3

In [26]: L.count(3)                 # Count how many 3 in L
Out[26]: 2

In [27]: L.reverse()                # Reverse L

In [28]: L
Out[28]: [3, 21, 21, 23, 2, 7, 3]

In [29]: L.sort()                   # Sort L in ascending order

In [30]: L
Out[30]: [2, 3, 3, 7, 21, 21, 23]
```

Fig 1.23 (click to enlarge)

Tuples

A ***tuple*** is a series of items within brackets (()), which is immutable. This means that the length of the tuple is fixed, and the elements are also fixed.

Tuple Creation:

Figure 1.24 shows how you can create a tuple:

```
In [1]: tup1 = (1, 2, 3)              # Create a tuple variable called tup1

In [2]: tup1
Out[2]: (1, 2, 3)

In [3]: tup2 = 1, 2, 3               # Equavalent to tup1 = (1, 2, 3)

In [4]: tup2
Out[4]: (1, 2, 3)

In [5]: type(tup1)
Out[5]: tuple

In [6]: type(tup2)
Out[6]: tuple
```

Fig 1.24 (click to enlarge)

Items in a tuple can also be tuples. The nested tup created in Figure 1.25 contains two tuple: (4,5, and 6) and (7, and 8):

```
In [7]: nested_tup = (4, 5, 6), (7, 8)   # Create a tuple variable called
nested_tup

In [8]: nested_tup
Out[8]: ((4, 5, 6), (7, 8))
```

Fig 1.25 (click to enlarge)

If the *x* is iterable, the built-in function *tuple(x)* returns a tuple whose items are the same as the itmes in *x*. *tuple()* will create any empty tuple. Figure 1.26 shows how to create a tuple from a string or a list:

```
In [9]: tuple('hello')
Out[9]: ('h', 'e', 'l', 'l', 'o')

In [10]: tuple([2, 3, 4, 5])
Out[10]: (2, 3, 4, 5)
```

Fig 1.26 (click to enlarge)

Elements can be accessed similarly to lists. For example, you can get the value of the second item of tup1 (shown in Figure 1.27):

```
In [11]: tup1
Out[11]: (1, 2, 3)

In [12]: tup1[1]
Out[12]: 2
```

Fig 1.27 (click to enlarge)

Tuple Methods:

Since the length and items of a tuple cannot be modified it will only contain two built-in methods which are ***index*** and ***count***. Figure 1.28 is an example of how to use index and count:

```
In [13]: a = (1, 2, 2, 2, 3, 4, 2)

In [14]: a.count(2)                # Count how many 2 in a
Out[14]: 4

In [15]: a.index(2)                # Return the index of the first 2 in a
Out[15]: 1
```

Fig 1.28 (click to enlarge)

Sets

A ***set*** is an unordered collection of distinct items. It is often used to get unique values, membership testing, and mathematical set operations such as union, intersection, difference, etc.

Set Creation:

A set can be created by the set function or a list of items with curly braces({ }) (shown in Figure 1.29):

```
In [16]: set([2, 2, 2, 1, 3, 3])
Out[16]: {1, 2, 3}

In [17]: {2, 2, 2, 1, 3, 3}
Out[17]: {1, 2, 3}
```

Fig 1.29 (click to enlarge)

From the outputs, we can see that it removes duplicated elements automatically.

Set Methods:

Please see Table 1.3.2 which shows a list of commonly used set methods.

Table 1.3.2: All Set Methods

Method	Description
s.copy()	Return a new set

Method	Description
s.difference(S1)	**Returns the set of all items of *S* that aren't in *S1***
s.intersection(S1)	**Returns the set of all items of *S* that are also in *S1***
S.issubset(S1)	**Returns True if all items of *S* are also in *S1*; otherwise, returns False**
S.issuperset(S1)	**Returns True if all items of *S1* are also in *S*; otherwise, returns False**
S.symmetric_difference(S1)	Update the set, keeping only elements found in either set, but not in both
S.union(S1)	Returns the set of all items of S and S1
S.add(x)	Add element *elem* to the set
S.clear()	Remove all elements from the set
S.discard()	Remove element *elem* from the set if it is present
S.pop()	Remove and return an arbitrary element from the set. Raises KeyError if the set is empty
S.Remove(x)	Remove an element from the set. Raises KeyError if element is not contained in the set

Figure 1.30 shows an example of how to get a union or an intersection of two sets, *a* and *b.* The union and the intersection can also be achieved by using | or & operators:

```
In [18]: a = {1, 2, 3, 4, 5}

In [19]: b = {3, 4, 5, 6, 7, 8}

In [20]: a.union(b)
Out[20]: {1, 2, 3, 4, 5, 6, 7, 8}

In [21]: a | b
Out[21]: {1, 2, 3, 4, 5, 6, 7, 8}

In [22]: a.intersection(b)
Out[22]: {3, 4, 5}

In [23]: a & b
Out[23]: {3, 4, 5}
```

Fig 1.30 (click to enlarge)

Dictionary

dict is the most important built-in data structure in Python as it enables faster searches. It it also referring to hashmap or associative array in other programming languages.

dict Creation:

dict is a collection of key-value pairs, which is indexed by keys instead of integer indexes like lists or tuples. Keys should be immutable data types such as numbers and strings. Tuples can also be used as keys if it only contains numbers, strings, or tuples. dict itself is mutable.

One way for creating dict is to use a sequence of key-value pairs within curly braces {} and colons to separate keys and values in pairs:

```
In [24]: {'x':34, 'y':12, 'z':7}   # or dict(x = 34, y = 12, z = 7)
Out[24]: {'x': 34, 'y': 12, 'z': 7}

In [25]: {1 :2, 2 : 3}             # or dict([1, 2], [2, 3])
Out[25]: {1: 2, 2: 3}

In [26]: {}                        # or use dict() to creat a empty dictionary
Out[26]: {}
```

Fig 1.31 (click to enlarge)

You can access, add, or modify elements using the same method as list or tuple, except that the indexes are keys instead of integers (see Figure 1.32):

```
In [27]: d = {'x':34, 'y':12, 'z':7}

In [28]: d['x']                  # Return the value for the key 'x'
Out[28]: 34

In [29]: d['y'] = 0              # Change the value for the key 'y'

In [30]: d
Out[30]: {'x': 34, 'y': 0, 'z': 7}

In [31]: d['a'] = 78             # Insert 'a': 78 into d

In [32]: d
Out[32]: {'x': 34, 'y': 0, 'z': 7, 'a': 78}
```

Fig 1.32 (click to enlarge)

Dictionary Method:

Below you can view Table 1.3.3 for all dictionary methods

Table 1.3.3: All Dictionary Methods

Method	Description
Key in D	Return True if *d* has a key *key*, else False
Key not in D	Equivalent to not key in d.
iter(D)	Return an iterator over the keys of the dictionary. This is a shortcut for iter(d.keys()).
D.copy()	Return a shallow copy of the dictionary.
D.items	Return a new view of the dictionary's items ((key, value) pairs)
D.keys()	Return a new view of the dictionary's keys.
D.values()	Return a new view of the dictionary's values.
D.get(k[,x])	Return the value for *key* if *key* is in the dictionary, else *default*. If *default* is not given, it defaults to None, so that this method never raises a KeyError.
D.clear()	Remove all items from the dictionary.
D.update(D1)	Update the dictionary with the key/value pairs from *other*, overwriting existing keys.

Method	Description
D.setdefault(k[, x])	If *key* is in the dictionary, return its value. If not, insert *key* with a value of *default* and return *default*. *default* defaults to None.
D.pop(k[, x])	If *key* is in the dictionary, remove it and return its value, else return *default*. If *default* is not given and *key* is not in the dictionary, a KeyError is raised.
D.popitem()	Remove and return an arbitrary (key, value) pair from the dictionary.

Below you can find examples from Table 1.3.3

```
In [33]: d
Out[33]: {'x': 34, 'y': 0, 'z': 7, 'a': 78}

In [34]: 0 in d
Out[34]: False

In [35]: d.items()
Out[35]: dict_items([('x', 34), ('y', 0), ('z', 7), ('a', 78)])

In [36]: d.keys()
Out[36]: dict_keys(['x', 'y', 'z', 'a'])

In [37]: d.values()
Out[37]: dict_values([34, 0, 7, 78])

In [38]: d.get('a')
Out[38]: 78

In [39]: d.pop('a')
Out[39]: 78

In [40]: d
Out[40]: {'x': 34, 'y': 0, 'z': 7}

In [41]: d.popitem()
Out[41]: ('z', 7)

In [42]: d
Out[42]: {'x': 34, 'y': 0}
```

Fig 1.33 (click to enlarge)

Lesson 1.4: Control Flow Statement

Python has built-in keywords to realize conditional logic, loops, and other control flow concepts

(click the tabs to learn more)

The *If* Statement:

The Python conditional logic is comprised of ***if***, ***elif***, and ***else*** statements. Here's the syntax for the *if* statement:

if *expression:statement(s)*

elif *expression: statement(s)*

elif *expression: statement(s)*

else: *statement(s)*

The ***elif*** and ***else*** statement are optional. Please be aware that Python does not have a switch statement like other programming languages.

Example

Figure 1.34 an example for an *if* statement:

```
In [1]: x = 4

In [2]: if x < 0:
   ...:     print('The input is negative')
   ...: elif x == 0:
   ...:     print('The input is equal to zero')
   ...: else:
   ...:     print('The input is positive')
The input is positive
```

Fig 1.34 (click to enlarge)

The *For* Statement:

The ***for*** statement is used to iterate over a collection of items such as a list, tuple, or dict. The syntax for the *for* statement is:

For *iter in iteration: statement(s)*

Examples

Example 1: Figure 1.35 shows an example of using a ***for*** loop to calculate the sum of a list:

```
In [3]: L1 = [3, 6, 2, 8, 78]

In [4]: total = 0

In [5]: for val in L1:
   ...:     total += val                        # calculate the sum of L1

In [6]: total
Out[6]: 97

In [7]: sum(L1)
Out[7]: 97
```

Fig 1.35 (click to enlarge)

Example 2: Iterate a dict

Figure 1.36 is an example of using a ***for*** loop to print out key-value pairs whose key or value is *None*:

```
In [8]: d = {'a': 3, 'b': 1, 'c': None, 'd': 6, None: '3'}

In [9]: for key, value in d.items():
   ...:     if not key or not value:          # If key or value is None
   ...:         print(key, value)             # Print out key - value pairs
c None
None 3
```

Fig 1.36 (click to enlarge)

The *While* Statement:

The ***while*** statement executes its block until the logic statement becomes False:

The syntax for the while statement is: while *logic statement: statement(s)*

Example

Figure 1.37 is an example for a while statement:

```
In [10]: L = [3, 6, 2, 8, 78]

In [11]: n = len(L)

In [12]: i = 0

In [13]: while i < n:
    ...:     print(L[i])
    ...:     i += 1
3
6
2
8
78
```

Fig 1.37 (click to enlarge)

The Break Statement:

The ***break*** statement is used to terminate a *for* or *while* loop earlier if some condition is True.

Example

Fig 1.38 is an example of using a loop to calculate the summation of a list.

```
In [14]: L2 = [11, 3, 135, None, 9]

In [15]: result = 0

In [16]: for i in range(len(L2)):
    ...:     if L2[i] is None:
    ...:         print('L2[', i, '] is not a valid number')
    ...:         break
    ...:     result += L2[i]
L2[ 3 ] is not a valid number
```

Fig 1.38 (click to enlarge)

The Continue Statement:

Instead of terminating a *for* or *while* loop like break statement, the ***continue*** statement will only skip the current iteration and continue to next iteration.

Example

Figure 1.39 is an example of using a *for* loop to calculate the summation of a list. By using the *continue* statement, calculation will skip the *None* element and continue adding elements to the *result*:

```
In [17]: result = 0

In [18]: for value in L2:
    ...:     if value is None:
    ...:         continue
    ...:     result += value

In [19]: result
Out[19]: 158
```

Fig 1.39 (click to enlarge)

The Pass Statement

If you have not action to perform, a ***pass*** statement can be used to fill in the statement requirement for the body but it will 'pass' on that action.

Example

Fig 1.40 shows an example of using a *for* loop to calculate the summation of a list. The for loop will skip *None* values and continue calculations:

```
In [20]: result = 0

In [21]: for value in L2:
    ...:     if value is None:
    ...:         pass
    ...:     else:
    ...:         result += value

In [22]: result
Out[22]: 158
```

Fig 1.40 (click to enlarge)

Lesson 1.5: Functions (6 of 7)

Lesson 1.5: Functions

(click tabs to learn more)

The *Def* Statement

The *def* statement is used to define a function. The syntax of a def statement is:

def *function name (parameter1, parameter2,...):statements*

Parameters are optional in function. If you have multiple parameters, they should be separated by commas (,).

Example

Fig 1.41 is an example to define a summation function *my_add*:

```
In [23]: def my_add(x, y):
    ...:     x + y  # this function doesn't return anything

In [24]: my_add(1, 5)
```

Fig 1.14 (click to enlarge)

There is no output after typing *my_add* (1,5), since this function doesn't have a return statement.

The *Return* Statement

The *return* statement is used to return a result from a function. If *return* executes, the function will terminate.

Example

Fig 1.42 is an example of defining the multiplication function *multiply*:

```
In [25]: def multiply(x, y):
    ...:     return x*y

In [26]: multiply(4, 8)
Out[26]: 32
```

Fig 1.42 (click to enlarge)

Please refer to this link (https://docs.python.org/3/library/functions.html) about built-in Python functions.

Lesson 1 References

- Chapter 2, Python for Data Analysis, Wes Makinney, 2nd edition.
- Python tutorial https://docs.python.org/3/tutorial/datastructures.html (https://docs.python.org/3/tutorial/datastructures.html)
- Chapter 4, Python in a Nutshell, Alex Martelli, 2nd edition.