Lecture 6 – Data Types , Values and Variables in Python

Understanding Data Types

- In Python, like in all programming languages, data types are used to classify one particular type of data.

- This is important because the specific data type you use will determine what values you can assign to it and what you can do to it (including what operations you can perform on it).

Numbers

- Any number you enter in Python will be interpreted as a number; **you are not required to declare** what kind of data type you are entering.

- Python will consider any number written **without decimals as an integer** (as in 138) and any **number written with decimals as a float** (as in 138.0).

Integers

- Like in math, integers in computer programming are whole numbers that can be positive, negative, or 0 (…, -1, 0, 1, …).

- An integer can also be known as an int.

- As with other programming languages, you should not use commas in numbers of four digits or more, so when you write 1,000 in your program, write it as 1000.

- We can print out an integer in a simple way like this:

```
>>>
print (-168)

-168
>>>
```

- Or, we can **declare a variable**, which in this case is essentially a symbol of the number we are using or manipulating, like :

```
my_int = -25
print(my_int)

-25
>>>
```

Floating-Point Numbers

- A floating-point number or a float is a real number, meaning that it can be either a **rational** or an **irrational number**.

- Because of this, floating-point **numbers** can be numbers **that** can **contain a fractional part**, such as 9.0 or -116.42.

- Simply speaking, for the purposes of thinking of a float in a Python program, it is a number that contains a decimal point

```
print (17.5)
17.5
>>>
```

- We can also declare a variable that stands in for a float, like :

```
 my_flt = 17.5
print (my_flt)
17.5
>>>
```

Booleans

- The Boolean data type can be one of two values, either True or False.

```
print ( 9 > 6 )
True
>>>
```

- We can also declare a variable that stands in for a Boolean , like :

```
my_bool = 5 > 8
print(my_bool)

False
>>>
```

Strings

- A string is a sequence of one or more characters (letters, numbers, symbols) that can be either a constant or a variable.

- Strings exist within either **single quotes ' or double quotes "** in Python.

- So, to create a string, enclose a sequence of characters in quotes.

```
print('This is a string in single quotes.')

This is a string in single quotes.
 >>>


print("This is a string in double quotes.")

This is a string in double quotes.
>>>
```

- As with other data types, we can store strings in variables

```
hw = "Hello, World!"
print(hw)


Hello, World!
>>>
```

Lists

- A list is a mutable, or changeable, ordered sequence of elements.

- **Each element** or value that is inside of a list is **called an item**.

- Just as strings are defined as characters between quotes, lists are defined by having values between square brackets **[ ].**

A **list of integers**:
[-3, -2, -1, 0, 1, 2, 3]

A **list of floats**:
[3.14, 9.23, 111.11, 312.12, 1.05]

A **list of strings**:
['shark', 'cuttlefish', 'squid', 'mantis shrimp']

If we **define** our string list as sea_creatures variable:

```
sea_creatures = ['shark', 'cuttlefish', 'squid', 'mantis shrimp']
```

We can print them out by calling the variable:

```
print(sea_creatures)
```

```
['shark', 'cuttlefish', 'squid', 'mantis shrimp']
>>>
```

Tuples

- A tuple is used for **grouping data**. **It is an or unchangeable, ordered sequence of elements.**

- Tuples are very similar to lists, but they **use parentheses ( ) instead of square brackets** and because they are unchangeable. **Their values cannot be modified.**

- A tuple looks like this:

  ('blue coral', 'staghorn coral', 'pillar coral'**)**

  We can **store** a tuple in a **variable** and print it out:

```
coral = ('blue coral', 'staghorn coral', 'pillar coral')

print(coral)


('blue coral', 'staghorn coral', 'pillar coral')
>>>
```

Dictionaries

- The dictionary is Python's built-in mapping type. This means that dictionaries **map keys to values** and these key-value pairs are a useful way to store data in Python.

- A dictionary is constructed with curly braces on either side **{ }.**

- Typically used to **hold data** that are related.

a dictionary looks like this:

{'name': 'Sammy', 'animal': 'shark', 'color': 'blue', 'location': 'ocean'**}**

The words to the **left of the colons are the keys**. Keys can be made up of any unchangeable data type.

The **keys** in the dictionary above **are**: 'name', 'animal', 'color', 'location'.

The words to the **right of the colons are the values**. Values can be comprised of any data type.

The **values** in the dictionary above are: 'Sammy', 'shark', 'blue', 'ocean'.

Like the other data types, let's **store the dictionary inside a variable, and print it out:**

```
sammy = {'name': 'Sammy', 'animal': 'shark', 'color': 'blue', 'location': 'ocean'}

print(sammy)
```

```
{'name': 'Sammy', 'animal': 'shark', 'color': 'blue', 'location': 'ocean'}
>>>
```

If we want to isolate Sammy's color, we can do so by calling sammy['color'] and print that out

```
print(sammy['color'])

blue
>>>

print(sammy['animal'])

shark
>>>


print(sammy['location'])

ocean
>>>
```

Variables in Python

The variable name is used to reference a stored value within a computer program.

my_int = 103204934813

- The variable name (my_int)
- The assignment operator, also known as the equal sign (=)
- The value that is being tied to the variable name (103204934813)

As soon as we set my_int equal to the value of 103204934813, we can use my_int in the place of the integer.

print(my_int)

Output will be 103204934813

Variables can represent any data type, not just integers:

```
my_string = 'Hello, World!'
my_flt = 45.06
my_bool = 5 > 9
#A Boolean value will return either True or False
my_list = ['item_1', 'item_2', 'item_3', 'item_4']
my_tuple = ('one', 'two', 'three')
my_dict = {'letter': 'g', 'number': 'seven', 'symbol': '&'}
```

If you print any of the above variables, Python will return what that variable is equivalent to.

Example :

```
print(my_string)
 print(my_flt)
 print(my_bool)
 print(my_list)
 print(my_tuple)
 print(my_dict)
```

Sample Run

```
Hello, World!
45.06
False
['item_1', 'item_2', 'item_3', 'item_4']
('one', 'two', 'three')
{'letter': 'g', 'number': 'seven', 'symbol': '&'}
```

Naming Variables: Rules and Style

The naming of variables is quite flexible, but there are some rules you need to keep in mind:

- Variable names must only be one word (as in no spaces)
- Variable names must be made up of only letters, numbers, and underscore (_)
- Variable names cannot begin with a number.

Example

VALID	INVALID	WHY INVALID
my_int	my-int	Hyphens are not permitted
int4	4int	Cannot begin with a number
MY_INT	$MY_INT	Cannot use symbols other than _
another_int	another int	Cannot be more than one word

Some notes about style.

CONVENTIONAL STYLE	UNCONVENTIONAL STYLE	WHY UNCONVENTIONAL
my_int	myInt	camelCase not conventional
int4	Int4	Upper-case first letter not conventional
my_first_string	myFirstString	camelCase not conventional

Reassigning Variables

As the word variable implies, Python variables can be changed.

```
#Assign x to be an integer
x = 76
print(x)

#Reassign x to be a string
x = "Sammy"
print(x)
```

```
76
Sammy
>>>
```

Multiple Assignment

With Python, you **can assign one single value to several variables at the same time.**

This lets you initialize several variables at once, which you can reassign later in the program yourself, or through user input.

Through multiple assignment, you can set the variables x, y, and z to the value of the integer 0:

Example

```
x = y = z = 0
print(x)
print(y)
print(z)
```

```
0
0
0

>>>
```

Python also allows you to assign several values to several variables within the same line.

Each of these values can be of a different data type

```
j, k, l = "shark", 2.05, 15
print(j)
print(k)
print(l)
```

```
shark
2.05
15

>>>
```

Global and Local Variables

When using variables within a program, it is important to keep variable scope in mind. A variable's **scope refers to the particular places it is accessible within the code of a given program**. This is to say that not all variables are accessible from all parts of a given program — some variables will be **global and some will be local.**

How To Convert Data Types

- In Python, **data types** are used to classify **one particular type of data**, determining the values that you can assign to the type and the operations you can perform on it.

- When programming, there are times **we need to convert values between** types in order to manipulate values in a different way.

- For example, we may need to **concatenate numeric values with strings**, or **represent decimal places in numbers that were initialized as integer values.**

Converting Number Types In Python,

- There are two number data types: **integers and floating-point** numbers or floats.

- Python has **built-in methods** to allow you to easily convert integers to floats and floats to integers.

Converting Integers to Floats

```
float(57)
```

```
57.0
>>>
```

Or

```
f = 57
print(float(f))
```

```
57.0
>>>
```

Converting Floats to Integer

int(390.8)

390
>>>

Or

```
b = 125.0
c = 390.8
print(int(b))
print(int(c))
```

125
390
>>>

Converting Strings to Numbers

Strings can be converted to numbers by using the int() and float() methods.

```
# Here age is a string object
age = "18"
print(age)
# Converting string to integer
int_age = int(age)
print(int_age)
float_age = float(age)
print(float_age)
```

```
18
18
18.0
>>>
```

More about Strings

Recall that Strings exist within either **single quotes ' or double quotes "** in Python, so to create a string, enclose a sequence of characters in quotes.

Example

```
print("Let's print out this string.")

print('This is Also a String.')
```

Escape Characters

An *escape character* lets you use characters that are otherwise impossible to put into a string. An escape character consists of a backslash **(\)** followed by the character you want to add to the string.

For example, the escape character for a **single quote is \'**. You can use this inside a string that begins and ends with single quotes. To see how escape characters work, enter the following into the interactive shell:

Example

```
print("Let\'s print out this string.")

Let's print out this string.

>>>
```

The following are lists of the escape characters you can use

\' Single quote

\" Double quote

\n New Line

\t Tab

\\ Backslash

Example

```
print("Hello there!\nHow are you?\nI\'m \tdoing \tfine.")
```

Output

```
Hello there!
How are you?
I'm     doing      fine.
>>>
```

sep='*separator*' **Optional**.

In the print statement , you can specify how to separate the objects, if there are more than one. The default is ' '

Example

```
print("Hello", "how are you?", sep=" --- ")
```

Output

```
Hello --- how are you?
>>>
```

Example :

```
Numbers = [1,2,3]
Chars = ("A","B")
string = "Hi  Hello "

print(Numbers , Chars ,string, sep=" << .. >> ")
```

Output

```
[1, 2, 3] << .. >> ('A', 'B') << .. >> Hi  Hello
>>>
```

Raw Strings

A *raw string* **completely ignores all escape characters** and prints any backslash that appears in the string.

Example

```
print(r'That is Carol\'s cat.')
```

Output

```
That is Carol\'s cat.
>>>
```

Example

```
print('That is Carol\'s cat.')
```

Output

```
That is Carol's cat.
>>>
```

end parameter in print() statement

By default Python‘s print() function ends with a newline. How to print without a newline

Example

```
print("Welcome to", end = ' ')
print("Python Programming")
```

Output

```
Welcome to Python Programming
```

Example

```
print('A','B', sep='', end='')
print('G')

#\n provides new line after printing the year

print('2','1','2023', sep=' - ', end='\n')
print('white','Red', 'Blue' , sep=', ', end=' @ ')
print('Texas')
```

Output :

ABG
2 - 1 - 2023
white, Red, Blue @ Texas

Multiline Strings with Triple Quotes

A multiline string in Python **begins and ends with either three single quotes or three double quotes**. Any quotes, tabs, or newlines in between the “triple quotes” are considered part of the string. Python’s indentation rules for blocks do not apply to lines inside a multiline string.

Example

```
print('''Dear Alice,

Eve's cat has been arrested for no reason.

Sincerely,
Bob''')
```

Sample run

```
Dear Alice,

Eve's cat has been arrested for no reason.

Sincerely,
Bob
>>>
```

Useful String Methods

Several string methods analyze strings or create transformed string values.

The upper(), lower(), isupper(), and islower() Methods

The **upper() and lower()** string methods return a new string where all the letters in the original string have been **converted to uppercase or lowercase, respectively**.

Nonletter characters in the string remain unchanged. Enter the following into the interactive shell:

The `isupper()` method returns True if **all** the characters are in upper case, otherwise False

The `islower()` method returns True if **all** the characters are in lower case, otherwise False.

```
spam = 'Hello, world!'
spam = spam.upper()
print(spam)
spam = spam.lower()
print(spam)
```

```
HELLO, WORLD!
hello, world!
>>
```

From the command line type the following

```
>>> spam = 'Hello, world!'

>>> spam.islower()
False
>>> spam.isupper()
False
>>> 'HELLO'.isupper()
True
>>> 'abc12345'.islower()
True
>>> '12345'.islower()
False
>>> '12345'.isupper()
False
```

What is the output of the following

```
>>> 'Hello'.upper()

>>> 'Hello'.upper().lower()

>>> 'Hello'.upper().lower().upper()

>>> 'HELLO'.lower()

>>> 'HELLO'.lower().islower()
```

The isX() Methods

Along with islower() and isupper(), there are several other string methods that have names beginning with the word ***is***.

These methods **return a Boolean value** that describes the nature of the string.

Example :

- **isdigit()** Returns True if all the characters are digits, otherwise False.
-
- **isalpha()** Returns True if the string consists only of letters and isn't blank
- **isalnum()** Returns True if the string consists only of letters and numbers and is not blank
- **isdecimal()** Returns True if the string consists only of numeric characters and is not blank
- **isspace()** Returns True if the string consists only of spaces, tabs, and newlines and is not blank
- **istitle()** Returns **True** if the string consists only of words that begin with an uppercase letter followed by only lowercase letters

Enter the following into the interactive shell:

```
>>> 'hello'.isalpha()
True

>>> 'hello123'.isalpha()
False

>>> 'hello123'.isalnum()
True
>>> 'hello'.isalnum()
True

>>> '123'.isdecimal()
True

>>> '    '.isspace()
True

>>> 'This Is Title Case'.istitle()
True

>>> 'This Is Title Case 123'.istitle()
True

>>> 'This Is not Title Case'.istitle()
False

>>> 'This Is NOT Title Case Either'.istitle()
False
```

```
txt = "50800"
x = txt.isdigit()
print(x)
```

```
True
>>>
```

```
txt = "6089a"
x = txt.isdigit()
print(x)
```

```
False
>>>
```

Justifying Text with the rjust(), ljust(), and center() Methods

The rjust() and ljust() string methods return a padded version of the string they are called on, with spaces inserted to justify the text.

The first argument to both methods is an integer length for the justified string.

Example

```
>>> 'Hello'.rjust(10)
'     Hello'


>>> 'Hello'.rjust(20)
'               Hello'


>>> 'Hello, World'.rjust(20)
'         Hello, World'


>>> 'Hello'.ljust(10)
'Hello     '
```

An optional second argument to rjust() and ljust() will specify a fill character other than a space character. Enter the following into the interactive shell:

Example

```
>>> 'Hello'.rjust(20, '*')

'***************Hello'

>>> 'Hello'.ljust(20, '-')

'Hello---------------'
```

The **center()** string method works like ljust() and rjust() but centers the text rather than justifying it to the left or right. Enter the following into the interactive shell:

```
>>> 'Hello'.center(20)

'       Hello        '

>>> 'Hello'.center(20, '=')

'=======Hello========'
```

What is the output of the following :

```
movie = "2001: A SAMMY ODYSSEY"
book = "A Thousand Splendid Sharks"
poem = "sammy lived in a pretty how town"
print(movie.islower())
print(movie.isupper())
print(book.istitle())
print(book.isupper())
print(poem.istitle())
print(poem.islower())
```