Rakamin Academy

HOME CREDIT

Virtual Internship Experience

Programming Language Scripting

Data Types and Library Installation
CRUD Dataframe and Table, Merge, Export, and Import

1. Data Types and Library Installation

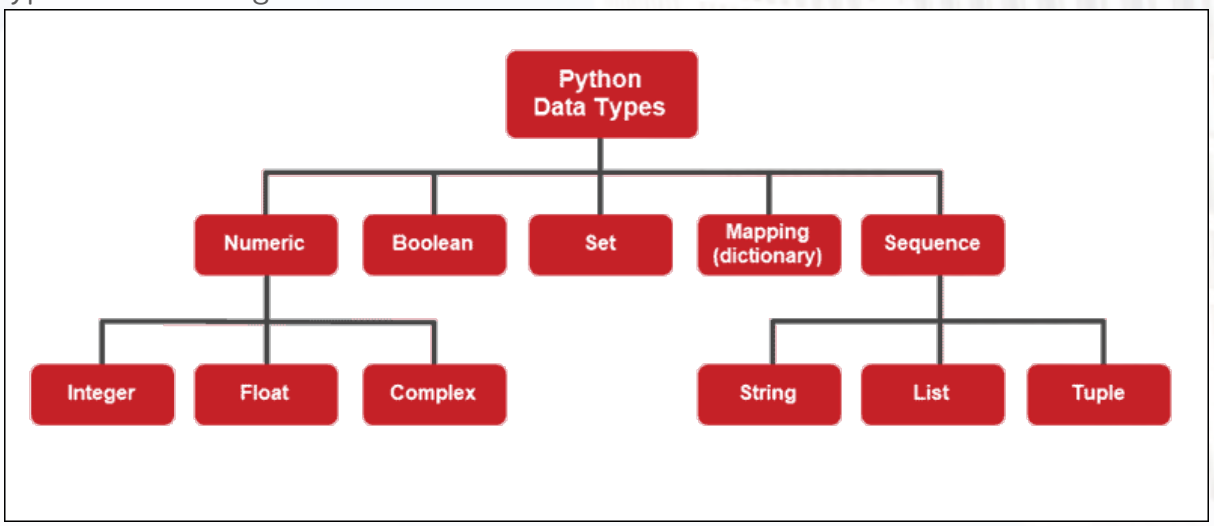
Basic Data Types in Python

A data type is a characteristic that tells the compiler (or interpreter) how a programmer intends to use the data. There are two general categories of data types, differing whether the data is changeable after definition:

1. Immutable. Data types that are not changeable after assignment.
2. Mutable. Data types that are changeable after assignment.

Note: Variable IDs change when changing data for immutable types, whereas mutable types retain the same variable ID after the change. Check the variable ID with the built-in function id(<variable>).

Variables store different types of data. Creating a variable of a particular data type creates an object of a data type class. The Python interpreter automatically assumes a type when creating a variable.



The data type of any object is found using the built-in type() function. The output shows the name of the class for the given object.

Numeric Type

Numeric type objects represent number values. They are divided into three subgroups:

- Integers

Integer values belong to the int class. Specifically, integers represent positive or negative whole numbers without a decimal. Some examples of integers include:

```
print(0, "is", type(0))
print(-1, "is", type(-1))
print(10, "is", type(10))
print(1234567890, "is", type(1234567890))
```

```
0 is <class 'int'>
-1 is <class 'int'>
10 is <class 'int'>
1234567890 is <class 'int'>
```

Integers have unlimited precision. There is no distinction between long and short numbers. The length depends on the computer memory:

```
very_long = 12345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345
print("Long numbers are type", type(very_long), ", such as this one:", very_long)
```

```
Long numbers are type <class 'int'> , such as this one: 12345678901234567890123456789012345678901234567
```

Binary, octal and hexadecimal number bases also evaluate to integers:

```
print(0b1010, "in base 2 is 1010 and the type is ", type(0b1010))
print(0o12, "in base 8 is 12 and the type is ", type(0o12))
print(0xA, "in base 16 is A and the type is ", type(0xA))
```

```
10 in base 2 is 1010 and the type is  <class 'int'>
10 in base 8 is 12 and the type is  <class 'int'>
10 in base 16 is A and the type is  <class 'int'>
```

- Floating Point Numbers

  Floating-point numbers in Python are numbers defined with a decimal point. The class type is float. For instance:

```
print(0.1, "is", type(0.1))
print(-1.0, "is", type(-1.0))
print(10., "is", type(10.))
print(123.4567890, "is", type(123.4567890))
```

```
0.1 is <class 'float'>
-1.0 is <class 'float'>
10.0 is <class 'float'>
123.456789 is <class 'float'>
```

Internally, they are binary fractions, which means the number approximates the decimal counterpart. The difference between the real and the approximated value is often unnoticeable. Rounding the value yields the exact number. For example:

```
1.2+2.2
```

```
3.4000000000000004
```

Alternatively, a number with the character E followed by a number indicates scientific notation:

```
1   print(1E2, "is", type(1E2))
2   print(1e2, "is", type(1e2))

100.0 is <class 'float'>
100.0 is <class 'float'>
```

Scientific notation belongs to the floating-point class of numbers as well. The syntax accepts both lowercase e as well as uppercase E.

Floating-point numbers after 1.79×10^{308} evaluate to infinity. The smallest non-zero number is 5.0×10-324. Smaller numbers evaluate to zero:

```
1   print(1.79e308, "is very large")
2   print(1.799e308, "is infinity")
3   print(5e-324, "is very small")
4   print(1e-324, "is zero")

1.79e+308 is very large
inf is infinity
5e-324 is very small
0.0 is zero
```

- Complex Numbers

  Complex numbers are often used in mathematical sciences. Python provides a class for complex numbers called complex. To write complex numbers, use:

```
1   print(1+2j, "is", type(1+2j))
2   print(2j, "is", type(2j))

(1+2j) is <class 'complex'>
2j is <class 'complex'>
```

Sequence Type

Sequence types help represent sequential data stored into a single variable. There are three types of sequences used in Python:

- String

  Strings are a sequence of bytes representing Unicode characters. The string type in Python is called str.

  Create a String:

  Depending on the use case and characters needed, there are four different ways to create a string. They differ based on the delimiters and whether a string is single or multiline.

  1. Create a string by using double quote delimiters:

```
print("This is a string with 'single' quotes delimited by double quotes")
print("Thanks to the \"\\\" character, we can use double quotes inside double quote delimiters")
```

```
This is a string with 'single' quotes delimited by double quotes
Thanks to the "\" character, we can use double quotes inside double quote delimiters
```

The example additionally shows how to display the backslash character in a string.

2. The example additionally shows how to display the backslash character in a string.

```
print('This is a string with "double" quotes delimited by single quotes')
print('Thanks to the \'\\\' character, we can use single quotes inside single quote delimiters')
```

```
This is a string with "double" quotes delimited by single quotes
Thanks to the '\' character, we can use single quotes inside single quote delimiters
```

There are two ways to delimit a multiline string.

a) Create a multiline string by using triple single quote delimiters:

```
print('''
"This is a multiline string with the following quotes:
'single'
"double"
"""triple double"""
Inside single double quotes
Delimited by triple single quotes"
''')
```

```
"This is a multiline string with the following quotes:
'single'
"double"
"""triple double"""
Inside single double quotes
Delimited by triple single quotes"
```

Use the triple single quotes to delimit a string if it contains both single, double, triple double quotes or ends in a double quote.

b) Create a string by using triple double quote delimiters:

```
print("""
'This is a multiline string with the following quotes:
'single'
"double"
```triple single```
Inside single quotes
Delimited by triple double quotes'
""")
```

```
'This is a multiline string with the following quotes:
'single'
"double"
```triple single```
Inside single quotes
Delimited by triple double quotes'
```

Use the triple double quote delimiter for strings containing single, double, triple single quotes or strings ending in a single quote.

Access Elements of a String

Strings in Python are arrays of characters. To access individual elements, use indexing:

```
s = "phoenixNap"
s[0], s[2], s[-1]
```

```
('p', 'o', 'p')
```

To access parts of a string, use slicing:

```
print(s[0:7])
print(s[-3:])
```

```
phoenix
Nap
```

Access the first element at index 0. Counting backward starting from -1 accesses the end of a sequence.

Since strings are arrays, you can loop through all the characters by using a for loop:

```
for letter in s:
    print(letter)
```

```
p
h
o
e
n
i
x
N
a
p
```

- List

A Python list is an ordered changeable array. Lists allow duplicate elements regardless of their type. Adding or removing members from a list allows changes after creation.

Create a List

Create a list in Python by using square brackets, separating individual elements with a comma:

```
A = [1, 2, "Bob", 3.4]
print(A, "is", type(A))
```

```
[1, 2, 'Bob', 3.4] is <class 'list'>
```

Make a nested list by adding a list to a list:

```
B = [A, 2, 3, 4]
print(B, "is", type(B))
```

```
[[1, 2, 'Bob', 3.4], 2, 3, 4] is <class 'list'>
```

Since Python lists are changeable, they allow creating empty lists and appending elements later, as well as adding or removing members to an existing list.

```
B = [A, 2, 3, 4]
print(B, "is", type(B))
```

```
[[1, 2, 'Bob', 3.4], 2, 3, 4] is <class 'list'>
```

Access Elements of a List

Lists are a sequence of elements. Access members by using indexing notation, where the first element is at index 0:

```
A[0], A[3], A[1]
```

```
(1, 3.4, 2)
```

Slicing a list returns all the elements between two indexes:

```
A[0:2]
```

```
[1, 2]
```

Negative indexes are also possible:

```
A[-1]
```

```
3.4
```

The -1 index prints the last element in the list. Negative indexing is especially useful for navigating to the end of a long list of members.

- Tuple

  Python Tuples are an array of unchangeable ordered elements. Once a tuple is stored into a variable, members cannot be added or removed. A tuple allows duplicate members of any type.

  Create a Tuple

  To create a tuple, use the standard round brackets, separating individual elements with a comma:

```
1  t = ("bare", "metal", "cloud", 2.0, "cloud")
2  print(t, "is", type(t))
```

```
('bare', 'metal', 'cloud', 2.0, 'cloud') is <class 'tuple'>
```

Make a nested tuple by adding a tuple to a tuple:

```
1  c = (t,"computing")
2  print(c, "is still", type(t))
```

```
(('bare', 'metal', 'cloud', 2.0, 'cloud'), 'computing') is still <class 'tuple'>
```

To create a tuple with a single element, use a comma after the first element:

```
1  p = ("phoenixNap")
2  n = ("phoenixNap",)
3  print("p is", type(p), "whereas n is", type(n))
```

```
p is <class 'str'> whereas n is <class 'tuple'>
```

Without a comma, the variable is a string.

Create an empty tuple using the round brackets without elements. Although it seems redundant since tuples are unchangeable, an empty tuple helps indicate a lack of data in certain use cases.

```
1  t[0], t[1], t[-1]
```

```
('bare', 'metal', 'cloud')
```

- Boolean Type

  Boolean data types belong to the bool class and determine the truth value of expressions. Objects of the Boolean type evaluate either to True or False:

```
1  print(type(True))
2  print(type(False))
```

```
<class 'bool'>
<class 'bool'>
```

- Set Type

  The Set data type is part of the set class. It stores data collections into a single variable. Sets are unordered and do not allow access to individual elements through indexing. Any duplicate values are ignored.

  To create a set, use the curly brackets notation and separate individual elements with a comma:

```
s = {1, 2, 3, 3, 3, 4}
print(s, "is", type(s))
```

```
{1, 2, 3, 4} is <class 'set'>
```

- Dictionary is a collection of data with key and value pairs belonging to the dict class. To create a dictionary, use the curly bracket notation and define the key value pairs. For example:

```
d = {"articles":10,
     "cost":2.2,
     True:"Okay!",
     2:"One"}
print(d, "is", type(d))
```

```
{'articles': 10, 'cost': 2.2, True: 'Okay!', 2: 'One'} is <class 'dict'>
```

The data type in Python is set automatically when writing a value to a variable. The class constructor for each data type allows setting the specific data type of a variable as well:

Data Type	Constructor
String	`str(<value>)`
Integer	`int(<value>)`
Floating-point	`float(<value>)`
Complex	`complex(<value>)`
List	`list((<value>, <value>))`
Tuple	`tuple((<value>, <value>))`
Boolean	`bool(<value)`
Set	`set((<value>, <value>))`
Dictionary	`dict((<key>=<value>, <key>=<value>))`

Library Installation

Python has developed a lot in recent years and its demand is also increased very much and what makes Python a very powerful language is its large and very effective powerful libraries. Such strong libraries give the power to python to do anything such as Web Crawling, Web Development, Machine Learning, Scientific Computing, Image Processing, Artificial Intelligence, Game Development, and many more so I am here with the top 10 Libraries of python which you must know about.

- Pandas: Pandas is a software library for python mainly for data manipulation and analysis. It has wide use in data science and statistics. It is based on DataFrame (a tabular data structure) for data manipulation with indexing. It provides several advanced data manipulation features such as Label-Based slicing, fancy indexing with support of multilevel indexing, Data set merging and joining, Reshaping and Pivoting of data sets, easy reading, and writing of data, Time Series, and many more. Highly recommended for Data Science enthusiasts.

- NumPy: In my opinion, this is one of the most important Python libraries. It provides advance Mathematic functionalities for scientific computing. It comes with the powerful and N-dimensional array object, broadcasting feature, vectorization support, etc.

- Matplotlib: A data scientist or a data analyzer can't live without this library. This provides the graphical representation of the data by plotting against the axis. Yes, a very powerful data visualization library. You can generate plots, scatter plots, histograms, power spectra, contours, etc. It is very similar to MATLAB and also resembles most of the MATLAB features and functionalities.

- TensorFlow: This is one of my favorite libraries. It is a framework that allows you to build Machine Learning models. It comes with a variety of tools to ease your implementation of Machine Learning. Companies such as Google, NVIDIA, Dropbox, etc use TensorFlow. It supports CUDA computing which allows you to train your model on GPU. TensorFlow is packed with some HIGH-LEVEL API such as Keras (API for building and training deep learning model), Estimators (API that provides models ready for large-scale training and production).

- OpenCV: It is a Computer Vision Library for mainly for image processing in real-time.

- PyGame: It is a cross-platform library for making multimedia applications such as Games.

- BeautifulSoup: It has applications while crawling and scraping the web when you need to extract data from HTML and XML. It parses the HTML and XML and extracts useful data.

- SciPy: With the three basic libraries out of the way, we come to SciPy, which is a scientific computing package that offers more mathematical functions than

NumPy. It actually uses NumPy to perform "various commonly used tasks in scientific programming, including linear algebra, integration (calculus), ordinary differential equation solving, and signal processing." The difference between NumPy and SciPy is that NumPy allows you to do basic sorting, indexing, and simple array mathematics to make sense of your data, and that is what you will use most of the time for practical uses. However, SciPy contains algebraic functions that is not fully contained in NumPy that is useful for more technical purposes.

- Keras : Similar to TensorFlow, Keras is another popular library that is used extensively for deep learning and neural network modules. Keras supports both the TensorFlow and Theano backends, so it is a good option if you don't want to dive into the details of TensorFlow.

- PyTorch: Next in the list of top python libraries for data science is PyTorch, which is a Python-based scientific computing package that uses the power of graphics processing units. PyTorch is one of the most commonly preferred deep learning research platforms built to provide maximum flexibility and speed.

2. CRUD Dataframe and Table, Merge, Export, and Import

CRUD Dataframe

The pandas series data structure enables us to perform CRUD operations, i.e. creating, reading, updating and deleting data. Once you perform these operations or a single operation on a series then a new series is returned.

- Creating Dataframe

  Pandas Series can be created in different ways from MySQL table, through excel worksheet (CSV) or from an array, dictionary, list etc. Let's look at how to create a series. Let's import Pandas first into the python file or notebook that you are working in:

```
import pandas as pd
ps = pd.Series([1,2,3,4,5])
print(ps)
```

  Output:

```
0 1
1 2
2 3
3 4
4 5
dtype: int64
```

- Reading Dataframe

  In order to read and select data from a series, you can use the index attribute by defining the index value or an index position (if no index is defined by you). Let's select data through an index value, for example:

```
ps = pd.Series([1,2,3,4,5], index=['a','b','c','d','e'])
print(ps['d'])
```

Output:

```
4
```

- Updating Dataframe

  You can update or replace the values in series as well by selecting the index position or value, for example:

```
ps = pd.Series([1,2,3,4,5], index=['a','d','c','d','e'])
ps['d'] = 900
print(ps)
```

Output:

```
a      1
d    900
c      3
d    900
e      5
dtype: int64
```

- Deleting Dataframe

  You can delete an entry in Series by selecting the del statement. Example:

```
ps = pd.Series([1,2,3,4,5], index=['a','b','c','d','e'])
del ps['a']
print(ps)
```

Output:

```
b    2
c    3
d    4
e    5
dtype: int64
```

Merge Dataframe

Merge DataFrame or named Series objects with a database-style join. A named Series object is treated as a DataFrame with a single named column. The join is done on columns or indexes. If joining columns on columns, the DataFrame indexes will be ignored. Otherwise if joining indexes on indexes or indexes on a column or columns, the index will be passed on. When performing a cross merge, no column specifications to merge on are allowed.

Syntax:

DataFrame.merge(*right, how='inner', on=None, left_on=None, right_on=None, left_index=False, right_index=False, sort=False, suffixes=('_x', '_y'), copy=True, indicator=False, validate=None*)

Parameter:

- Right: DataFrame or named Series Object to merge with.
- how{'left', 'right', 'outer', 'inner', 'cross'}, default 'inner'. Type of merge to be performed.
  - left: use only keys from left frame, similar to a SQL left outer join; preserve key order.
  - right: use only keys from right frame, similar to a SQL right outer join; preserve key order.
  - outer: use union of keys from both frames, similar to a SQL full outer join; sort keys lexicographically.
  - inner: use intersection of keys from both frames, similar to a SQL inner join; preserve the order of the left keys.
  - cross: creates the cartesian product from both frames, preserves the order of the left keys.

- On : label or list. Column or index level names to join on. These must be found in both DataFrames. If on is None and not merging on indexes then this defaults to the intersection of the columns in both DataFrames.
- left_on: label or list, or array-like. Column or index level names to join on in the left DataFrame. Can also be an array or list of arrays of the length of the left DataFrame. These arrays are treated as if they are columns.
- right_on: label or list, or array-like. Column or index level names to join on in the right DataFrame. Can also be an array or list of arrays of the length of the right DataFrame. These arrays are treated as if they are columns.
- left_index: bool, default False. Use the index from the left DataFrame as the join key(s). If it is a MultiIndex, the number of keys in the other DataFrame (either the index or a number of columns) must match the number of levels.
- right_index: bool, default False. Use the index from the right DataFrame as the join key. Same caveats as left_index.
- Sort: bool, default False. Sort the join keys lexicographically in the result DataFrame. If False, the order of the join keys depends on the join type (how keyword).
- Suffixes: list-like, default is ("_x", "_y"). A length-2 sequence where each element is optionally a string indicating the suffix to add to overlapping column names in left and right respectively. Pass a value of None instead of a string to indicate that the column name from left or right should be left as-is, with no suffix. At least one of the values must not be None.
- Copy: bool, default True. If False, avoid copy if possible.
- Indicator: bool or str, default False. If True, adds a column to the output DataFrame called "_merge" with information on the source of each row. The column can be given a different name by providing a string argument. The column will have a Categorical type with the value of "left_only" for observations whose merge key only appears in the left DataFrame, "right_only" for observations whose merge key only appears in the right DataFrame, and "both" if the observation's merge key is found in both DataFrames.
- Validate: str, optional. If specified, checks if merge is of specified type.
  - "one_to_one" or "1:1": check if merge keys are unique in both left and right datasets.
  - "one_to_many" or "1:m": check if merge keys are unique in left dataset.
  - "many_to_one" or "m:1": check if merge keys are unique in right dataset.
  - "many_to_many" or "m:m": allowed, but does not result in checks.

Import and Export Dataframe

After creating an intermediate or final dataset in pandas, we can export the values from the DataFrame to several other formats. The most common one is CSV, and the command to do so is df.to_csv('filename.csv'). Other formats, such as Parquet and JSON, are also supported.

After finishing our analysis, we may want to save our transformed dataset with all the corrections, so if we want to share this dataset or redo our analysis, we don't have to transform the dataset again. We can also include our analysis as part of a larger data pipeline or even use the prepared data in the analysis as input to a machine learning algorithm. We can accomplish data exporting our DataFrame to a file with the right format:

1. Import all the required libraries and read the data from the dataset using the following command:

```
import numpy as np
import pandas as pd

url = "https://opendata.socrata.com/api/views/cf4r-dfwe/rows.csv?accessType=DOWNLOAD"
df = pd.read_csv(url)
```

2. Export our transformed DataFrame, with the right values and columns, to the CSV format with the to_csv function. Exclude the index using index=False, use a semicolon as the separator sep=";", and encode the data as UTF-8 encoding="utf-8":

```
df.to_csv('radiation_clean.csv', index=False, sep=';', encoding='utf-8')
```

Reference :

1. https://phoenixnap.com/kb/python-data-types#ftoc-heading-9
2. https://tarunkr.medium.com/top-10-python-libraries-1934eaed433c
3. https://python-tricks.com/crud-in-series/
4. https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.merge.html
5. https://subscription.packtpub.com/book/data/9781789955286/1/ch01lvl1sec

07/exporting-data-from-pandas