PANDAS DATA TYPES

Regular Python does not have many data types. It only has string, float, binary, and complex numbers. There is no longer or short. There are no 32- or 64-bit numbers. Luckily, for most situations, this doesn't matter. It only matters when you require absolute precision or want to use the minimum amount of memory to store a value.

This matters when you are working with very large Pandas arrays since Pandas, if you recall, is limited by memory size. So, you would not want to use 64 bits when you only need 8.

(This tutorial is part of our Pandas Guide. Use the right-hand menu to navigate.)

Python numeric precision

Size and precision are different. But it's worth looking at precision in order to understand some of the limitations of Python. **Precision** means the number of decimal places.

For example, ⅓ in decimal format is the infinite series 0.3333333333333... The computer is not infinite, so at some point Python stops the decimal expansion and rounds that number off.

This behavior has some quirks.

When does zero equal zero?

Look at this code. It calculates the square root of 2 (or any other number) using a 3,000-year-old technique. The algorithm quits when the difference between two numbers is 0.

```
a=2
```

```
a1 = (a/2)+1
b1 = a/a1
aminus1 = a1
bminus1 = b1
while (aminus1-bminus1 != 0):
an = 0.5 * (aminus1 + bminus1)
bn = a / an
aminus1 = an
bminus1 = bn
print(an,bn,an-bn)
```

produces:

```
1.5 1.3333333333333333 0.16666666666666674
1.4166666666666665 1.411764705882353 0.004901960784313486
1.4142156862745097 1.41421143847487 4.2477996395895445e-06
1.4142135623746899 1.4142135623715002 3.1896707497480747e-12
1.414213562373095 1.4142135623730951 -2.220446049250313e-16
```

But you can see, just above, that the difference is never 0. Instead the computer rounds off, apparently in this case, at 16 significant digits. You can see this—when it reaches the answer 1.414213562373095, the code keeps running. That's because the difference between the two values does not converge to 0 despite taking the arithmetic mean of the two values.

If you want to understand how this algorithm works check out this clip:

In case you do want precise control over the number of decimal places, you can use the Python class **Decimal**. This keeps that code from running in a continuous loop.

```
from decimal import *
getcontext().prec = 5
a=Decimal(2)
a1 = Decimal((a/2)+1)
b1 = Decimal(a/a1)
aminus1 = a1
bminus1 = b1
i=0
while (Decimal(aminus1-bminus1) != Decimal(0)):
an = Decimal(Decimal(0.5) * (aminus1 + bminus1))
bn = Decimal(a / an)
aminus1 = an
bminus1 = bn
i=i+1
print(i,an,bn,an-bn)
```

It loops three times and quits, settling on the value of 1.4142 as the square root of 2. We could add more decimal places to get more precision.

```
1 1.5 1.3333 0.1667
```

```
2 1.4166 1.4118 0.0048

3 1.4142 1.4142 0.0000
```

NumPy & Pandas numeric data types

NumPy goes much further than that. It provides a low-level interface to c-type numeric types. (In other words, those numbers that you could declare when writing code in the C language). While we won't discuss those here yet, it also gives different data types suitable for:

- Time series
- Intervals
- Dates
- Categorical data
- Etc.

Below is the complete list. On the left is the object name. On the right is the string alias.

Data Type	**Alias**
Int64Dtype, ...	'Int8', 'Int16', 'Int32', 'Int64', 'UInt8', 'UInt16', 'UInt32', 'UInt64'
IntervalDtype	'interval', 'Interval', 'Interval', 'Interval]', 'Interval]'
DatetimeTZDtype	'datetime64'
CategoricalDtype	'category'
PeriodDtype	'period', 'Period'
SparseDtype	'Sparse', 'Sparse', 'Sparse'
BooleanDtype	'boolean'
StringDtype	'string'
none of the above	'object'

Integers & Floats

Here, for example, we have a date, float, boolean, and integer. We can let Pandas pick a scale for each numeric type. Or we can give that explicitly. If you don't give it explicitly Pandas either picks one or uses the generic **object**.

```
import pandas as pd
import datetime
df= pd.DataFrame({
"a": datetime.datetime(2020,12,14),
"b": 1.000003,
"c": True,
"d": 3
}, index=)
```

Ask Pandas for the data types:

```
df.dtypes
```

You can see it chooses 64 bits to store 1.000003 and 3. You only need 2 bits to store the number 3, but there is no option for 2-bit numbers. So, we would use int8 and use 8 bits, if space was a concern.

```
a    datetime64
b           float64
c              bool
d             int64
dtype: object
```

Now, make a Pandas series of 4 integers and coerce it to an 8 bit number.

```
s=pd.Series(,index=).astype('int8')
```

Use **dtypes** to show the data types:

```
s.dtypes
```

Results in:

```
dtype('int8')
```

The string 'int8' is an alias. You can also assign the dtype using the Pandas object representation of that pd.Int64Dtype.

```
t = pd.Int64Dtype
pd.Series(, dtype=t)
```

Related reading

- BMC Machine Learning & Big Data Blog
- Pandas: How To Read CSV & JSON Files
- Python Development Tools: Your Python Starter Kit
- Snowflake Guide, a series of tutorials
- Enabling the Citizen Data Scientists