notebook_2_python_data

March 21, 2018

1 Python Data - Objects and Types

1.1 Objectives

- Introduce Python data types
- Describe object-oriented design
- Understand the basics of working with objects
- Use `dir`, `type`, and `help` to explore python data
- Introduce variables and variable assignment
- Learn to import from a module
- Understand python namespaces and the dot notation

1.2 Python data

- All data have a *type* and *value*
  - *type* is based off the underlying class
  - *value* is returned by the Python interpreter
- We can use the `type` function to discover data classes/types

```
In [1]: type("Hello World")
```

```
Out[1]: str
```

```
In [2]: type(17)
```

```
Out[2]: int
```

```
In [3]: type(2.5)
```

```
Out[3]: float
```

1.3 Function calls

- In the last example, we *called* the `type` function
- Syntax: func_name(argument(s))

```
In [4]: type("Hi") # One argument
```

```
Out[4]: str
```

```
In [5]: print(1,2,3) # Many arguments
```

```
1 2 3
```

1.4 More on strings

- There are a number of forms for a string

```
In [6]: type('a string')
```

```
Out[6]: str
```

```
In [7]: type("also a string")
```

```
Out[7]: str
```

```
In [8]: type("""A
             multi-line
             string""")
```

```
Out[8]: str
```

```
In [9]: type('''Another
             multi-line
             string''')
```

```
Out[9]: str
```

1.5 Other basic types

- Boolean
- None

```
In [10]: type(True)
```

```
Out[10]: bool
```

```
In [11]: type(False)
```

```
Out[11]: bool
```

```
In [12]: type(None)
```

```
Out[12]: NoneType
```

1.6 Collection types

- Python comes with nice collection types

```
In [13]: type([1,2,3])
```

```
Out[13]: list
```

```
In [14]: type((1,2,3))
```

```
Out[14]: tuple
```

```
In [15]: type({"one":1, "two":2, "three":3})
```

```
Out[15]: dict
```

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

```
Out[16]: set
```

1.7 Type conversion

- convert types with the following
  - `int`, `str`, `float`, `list`, `tuple`, `dict`, `set`

1.8 Integers

- Converts floats and strings to ints
  - Truncates floats (toward zero)
  - Only works on valid strings
- **Note** Integers are infinite precision

```
In [17]: int(3.14)
```

```
Out[17]: 3
```

```
In [19]: int(3.9999) # Doesn't round, truncates
```

```
Out[19]: 3
```

```
In [20]: int(-3.9999) # Truncates toward 0
```

```
Out[20]: -3
```

```
In [21]: int("2345")        # parse a string to produce an int
```

```
Out[21]: 2345
```

```
In [23]: int("23bottles") #Not a valid int
```

```
---------------------------------------------------------------------------

ValueError                                Traceback (most recent call last)

<ipython-input-23-b83bd6a5825f> in <module>()
----> 1 int("23bottles") #Not a valid int


ValueError: invalid literal for int() with base 10: '23bottles'
```

```
In [24]: float("123.45")

Out[24]: 123.45

In [25]: float(17)

Out[25]: 17.0

In [26]: str(17)

Out[26]: '17'

In [27]: str(123.45)

Out[27]: '123.45'
```

1.9 Variables and Assignment

- Variables *hold* python data
- Variables are *assigned* with the **assignment statement**
- Syntax: `variable_name = value`

```
In [31]: message = "What's up, Doc?"

         n = 17

         pi = 3.14159

In [32]: 17 = n # Order matters


      File "<ipython-input-32-f64a7b2cdb48>", line 1
    17 = n # Order matters
                          ^
SyntaxError: can't assign to literal
```

1.10 Accessing data

- Evaluate a variable to *see* the stored data

```
In [33]: message
```

```
Out[33]: "What's up, Doc?"
```

```
In [34]: n
```

```
Out[34]: 17
```

```
In [35]: pi
```

```
Out[35]: 3.14159
```

1.11 Variable types

- A variable has the same type as the stored data

```
In [36]: type(message)
```

```
Out[36]: str
```

```
In [37]: type(n)
```

```
Out[37]: int
```

```
In [38]: type(pi)
```

```
Out[38]: float
```

1.12 Importing from Modules

- Many python tools need to be imported
- Use the **import statement**
- **Basic syntax** `import math`

```
In [39]: import math
```

1.13 Working with modules

- Use **dot notation** to access elements
- Use `dir` function to explore

```
In [40]: math.pi
```

```
Out[40]: 3.141592653589793
```

```
In [41]: math.sqrt(3)
```

```
Out[41]: 1.7320508075688772
```

```
In [ ]: dir(math)
```

1.14 Think of modules as folders

1.15 Using `help`

- The help function gives more information
- Give is a name!
  - Not a function call

```
In [43]: help(math.sqrt)
```

```
Help on built-in function sqrt in module math:

sqrt(...)
    sqrt(x)

    Return the square root of x.
```

1.16 Importing directly into the main namespace

- Typing `math.` gets annoying
- Use `from math import pi` to get direct access
- Beware of shadowing!

```
In [44]: import math
         math.sqrt(math.pi) # Annoying!

Out[44]: 1.7724538509055159

In [45]: from math import pi, sqrt

In [46]: sqrt(pi)

Out[46]: 1.7724538509055159
```

1.17 Object Oriented Design

- Programming paradigm
- All Python data are objects
- Code is are organized in objects
  - **attributes** data/state
  - **methods** functions for transforming data
  - **Instantiate** initial creation

1.18 Example - Fraction data type

```
In [47]: from fractions import Fraction

         help(Fraction)
```

```
Help on class Fraction in module fractions:

class Fraction(numbers.Rational)
 |  This class implements rational numbers.
 |
 |  In the two-argument form of the constructor, Fraction(8, 6) will
 |  produce a rational number equivalent to 4/3. Both arguments must
 |  be Rational. The numerator defaults to 0 and the denominator
 |  defaults to 1 so that Fraction(3) == 3 and Fraction() == 0.
 |
 |  Fractions can also be constructed from:
 |
 |    - numeric strings similar to those accepted by the
 |      float constructor (for example, '-2.3' or '1e10')
 |
 |    - strings of the form '123/456'
 |
 |    - float and Decimal instances
 |
 |    - other Rational instances (including integers)
 |
 |  Method resolution order:
 |      Fraction
 |      numbers.Rational
 |      numbers.Real
 |      numbers.Complex
 |      numbers.Number
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __abs__(a)
 |      abs(a)
 |
 |  __add__(a, b)
 |      a + b
 |
 |  __bool__(a)
 |      a != 0
 |
 |  __ceil__(a)
 |      Will be math.ceil(a) in 3.0.
 |
```

```
|  __copy__(self)
|
|  __deepcopy__(self, memo)
|
|  __eq__(a, b)
|      a == b
|
|  __floor__(a)
|      Will be math.floor(a) in 3.0.
|
|  __floordiv__(a, b)
|      a // b
|
|  __ge__(a, b)
|      a >= b
|
|  __gt__(a, b)
|      a > b
|
|  __hash__(self)
|      hash(self)
|
|  __le__(a, b)
|      a <= b
|
|  __lt__(a, b)
|      a < b
|
|  __mod__(a, b)
|      a % b
|
|  __mul__(a, b)
|      a * b
|
|  __neg__(a)
|      -a
|
|  __pos__(a)
|      +a: Coerces a subclass instance to Fraction
|
|  __pow__(a, b)
|      a ** b
|
|      If b is not an integer, the result will be a float or complex
|      since roots are generally irrational. If b is an integer, the
|      result will be rational.
|
|  __radd__(b, a)
```

```
 |      a + b
 |
 |  __reduce__(self)
 |      helper for pickle
 |
 |  __repr__(self)
 |      repr(self)
 |
 |  __rfloordiv__(b, a)
 |      a // b
 |
 |  __rmod__(b, a)
 |      a % b
 |
 |  __rmul__(b, a)
 |      a * b
 |
 |  __round__(self, ndigits=None)
 |      Will be round(self, ndigits) in 3.0.
 |
 |      Rounds half toward even.
 |
 |  __rpow__(b, a)
 |      a ** b
 |
 |  __rsub__(b, a)
 |      a - b
 |
 |  __rtruediv__(b, a)
 |      a / b
 |
 |  __str__(self)
 |      str(self)
 |
 |  __sub__(a, b)
 |      a - b
 |
 |  __truediv__(a, b)
 |      a / b
 |
 |  __trunc__(a)
 |      trunc(a)
 |
 |  limit_denominator(self, max_denominator=1000000)
 |      Closest Fraction to self with denominator at most max_denominator.
 |
 |      >>> Fraction('3.141592653589793').limit_denominator(10)
 |      Fraction(22, 7)
```

```
 |      >>> Fraction('3.141592653589793').limit_denominator(100)
 |      Fraction(311, 99)
 |      >>> Fraction(4321, 8765).limit_denominator(10000)
 |      Fraction(4321, 8765)
 |
 |  ----------------------------------------------------------------------
 |  Class methods defined here:
 |
 |  from_decimal(dec) from abc.ABCMeta
 |      Converts a finite Decimal instance to a rational number, exactly.
 |
 |  from_float(f) from abc.ABCMeta
 |      Converts a finite float to a rational number, exactly.
 |
 |      Beware that Fraction.from_float(0.3) != Fraction(3, 10).
 |
 |  ----------------------------------------------------------------------
 |  Static methods defined here:
 |
 |  __new__(cls, numerator=0, denominator=None, _normalize=True)
 |      Constructs a Rational.
 |
 |      Takes a string like '3/2' or '1.5', another Rational instance, a
 |      numerator/denominator pair, or a float.
 |
 |      Examples
 |      --------
 |
 |      >>> Fraction(10, -8)
 |      Fraction(-5, 4)
 |      >>> Fraction(Fraction(1, 7), 5)
 |      Fraction(1, 35)
 |      >>> Fraction(Fraction(1, 7), Fraction(2, 3))
 |      Fraction(3, 14)
 |      >>> Fraction('314')
 |      Fraction(314, 1)
 |      >>> Fraction('-35/4')
 |      Fraction(-35, 4)
 |      >>> Fraction('3.1415') # conversion from numeric string
 |      Fraction(6283, 2000)
 |      >>> Fraction('-47e-2') # string may include a decimal exponent
 |      Fraction(-47, 100)
 |      >>> Fraction(1.47)  # direct construction from float (exact conversion)
 |      Fraction(6620291452234629, 4503599627370496)
 |      >>> Fraction(2.25)
 |      Fraction(9, 4)
 |      >>> Fraction(Decimal('1.47'))
 |      Fraction(147, 100)
```

```
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  denominator
 |
 |  numerator
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  __abstractmethods__ = frozenset()
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from numbers.Rational:
 |
 |  __float__(self)
 |      float(self) = self.numerator / self.denominator
 |
 |      It's important that this conversion use the integer's "true"
 |      division rather than casting one side to float before dividing
 |      so that ratios of huge integers convert without overflowing.
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from numbers.Real:
 |
 |  __complex__(self)
 |      complex(self) == complex(float(self), 0)
 |
 |  __divmod__(self, other)
 |      divmod(self, other): The pair (self // other, self % other).
 |
 |      Sometimes this can be computed faster than the pair of
 |      operations.
 |
 |  __rdivmod__(self, other)
 |      divmod(other, self): The pair (self // other, self % other).
 |
 |      Sometimes this can be computed faster than the pair of
 |      operations.
 |
 |  conjugate(self)
 |      Conjugate is a no-op for Reals.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from numbers.Real:
 |
 |  imag
```

```
 |      Real numbers have no imaginary component.
 |
 |  real
 |      Real numbers are their real component.
```

In [48]:
```python
f = Fraction(10,8) # Instantiate a fraction
f # evaluate the object
```

Out[48]:
```
Fraction(5, 4)
```

1.19 Classes and objects

- A **class** is the blueprint for an object
  - i.e. the code that defines the object
  - class == type
- An **object** is an instance of a class
  - i.e. live data
- We can have multiple instances of a class

In [49]:
```python
type(f) # type = class
```

Out[49]:
```
fractions.Fraction
```

In [50]:
```python
f = Fraction(2,3) # One instance
g = Fraction(1,4) # Another instance
g + f
```

Out[50]:
```
Fraction(11, 12)
```

In [51]:
```python
type(g + f)
```

Out[51]:
```
fractions.Fraction
```

1.20 Use `dir` and dot notation with objects

- `dir` lists all attributes/methods
- Ignore members starting with _
- Use dot notation to access members

In [ ]:
```python
dir(f)
```

In [53]:
```python
f.denominator #Attribute
```

Out[53]:
```
3
```

In [54]:
```python
f.conjugate() #Method
```

```
Out[54]: Fraction(2, 3)
```

```
In [55]: help(f.conjugate)
```

```
Help on method conjugate in module numbers:

conjugate() method of fractions.Fraction instance
    Conjugate is a no-op for Reals.
```