Fairfield University
DOLAN
School of Business

# Python for Analytics

## Standard Data Types

Python Library Docs

# Learning Objectives

- **Theory:** You should be able to explain ...
  - The Python data type hierarchy
  - Primitive vs. higher-order types
  - Various kinds of Boolean Expressions
  - `Decimal`, `Fraction`, `array`, `dict`, `date`, etc.
  - The use of Hash Functions
- **Skills:** You should know how to ...
  - Use (and chain) comparison operators
  - Create Decimal and Fractional numbers
  - Create and modify a dictionary

# “Data Type” vs “Data Structure”

A **Data Type** defines a set of requirements:

- Kinds of data to be *encapsulated* (contained)
- Operations to be *supported* (applicable)

A **Data Structure** is a particular **implementation** of a Data Type.

**There is no practical difference** (unless you happen to be building your own data structures). We will use the terms interchangeably in this course.

# The Type Hierarchy

A Family Tree of Data Types

# Primitive Data Types

The most fundamental of all data types, the ones that are actually used to implement ***everything*** else.

| | |
|---|---|
| bit | binary 0 or 1 (used internally) |
| byte | 8 bits (used internally) |
| `int` | integer number (standard lib) |
| `float` | floating point number (standard lib) |
| `str` | string of unicode characters (standard lib) |

# Python Data Types

| | **Numbers** | **Immutable Sequences** | **Mutable Sequences** | **Sets** | **Mappings** | **Misc/Other** |
|---|---|---|---|---|---|---|
| Built-in Types | int, float,<br>complex | `str`, `tuple`,<br>`range`,<br>`bytes` | `list`,<br>`bytearray` | `set`,<br>`frozenset` | `dict` | `None`,<br>`Ellipsis`,<br>`Class`<br>etc. |
| Specialized Types | `Decimal`,<br>`Fraction` | `namedTuple` | `array`,<br>`deque` | `enum` | `Ordered Dict`,<br>`Chain Map` | `date`,<br>`time`,<br>`datetime`,<br>`Calendar`,<br>`heap` |

For more, RTFM sections 4, 8, and 9 of the Python Standard Library docs

# Boolean Expressions

The many ways of expressing True and False

# Always True … unless False

A boolean expression evaluates to True if it does not evaluate to False (yes, seriously).

The following all evaluate to False:

- Constants: None and False
- Numbers: 0, 0.0, Decimal(0), Fraction(0,1)
- Empty Sequences: ‘ ’, ( ), [ ], set(), range(0)

So, for example, the number 10 evaluates to True

# Truth Testing with bool()

We can evaluate any expression using the bool() conversion function.

```
bool(False)     → False
bool(True)      → True
bool(0.0)       → False
bool(10)        → True
bool(“False”)   → True
bool(())        → False
bool([10,20])   → True
```

# Comparison Operations

| | |
|---|---|
| < | Strictly less than |
| <= | Less than or equal |
| > | Strictly greater than |
| >= | Greater than or equal |
| == | Equal |
| != | Not Equal |
| is | Object identity (===) |
| is not | Negated object identity (!==) |

**Comparisons can be chained**

```
x < y <= z
x is not y > z
```

Evaluation is from *left to right*, trying each of the comparisons in order. If any of the comparisons fail, then the expression is `False`. Otherwise, it is `True`.

# Numeric Types

`int`, `float`, `complex`, and higher order types

# `int`, `float`, and `complex`

Reminder: **numbers** and **strings** are **immutable** in Python

We've already seen integer and floating point numbers, but complex numbers are new.

A `complex` number combines two `floats`:

- A *real* part
- An *imaginary* part

We likely won't see complex numbers in this class (or any other class, for that matter)

# Decimal Numbers

The `decimal` module provides the `Decimal` class for handling fixed-precision floating point arithmetic:

```
from decimal import *
getcontext().prec = 6  # set precision
Decimal(1) / Decimal(7)
   → Decimal('0.142857')
getcontext().prec = 28
Decimal(1) / Decimal(7)
   → Decimal('0.1428571428571428571428571429')
```

# Fractional Numbers

The `fractions` module provides the `Fraction` class with constructor `Fraction(numerator, denominator)`

```
from fractions import Fraction
Fraction(16,-10)
   → Fraction(-8,5)
Fraction(1,2)*Fraction(2,3)
   → Fraction(1,3)
```

# Sequences
## (Ordered Collections)

`tuple`, `range`, `str`, `list`, etc.

# Immutable Sequences

We’ve already seen `tuple`, `str`, and `range`

- All are immutable and cannot be changed once created

Of the three, tuples are the most general:

- A string is a sequence of characters
- A range is a sequence of consecutive numbers
- A tuple is a sequence of immutable items (characters, numbers, tuples, etc.)

# Mutable Sequences

Lists are the most common mutable sequences. You may also run across the `array`, which is like a list but with all the items inside of the same basic type

```
from array import array
array(‘i’,1,2)               # list of ints
array(‘f’,1.0,5.0,7.8)       # list of floats
array(‘u’,‘a’,‘↑’,‘☺’)       # list of characters
```

# Making Mutables Immutable

```
lst   = [3,4]      # lst is a (mutable) list
tpl =(‘1’,2,lst)
tpl              → (‘1’,2,[3,4])

lst.append(5)      # modifying the value of lst
lst              → [3,4,5]
tpl              → (‘1’,2,[3,4])
```

# Sets and Mappings
## (Unordered Collections)

`set`, `frozenset`, and `dict`

# A Digression About Hashing

A **hash function** converts arbitrary data (numbers, strings, etc.) to a digest of fixed length.

For example, let’s use the `md5` function on ‘Go Stags!’:

```
import hashlib
hashlib.md5(b‘Go Stags!’).hexdigest()
```

→ '59a060123aeddcba30023c46396aa5d8'

While digests are not guaranteed to be unique for every possible data, odds of duplicates are *very low*.

# Sets

A `set` is a collection of ***distinct hashable*** items

- Hashing is used to speed up the uniqueness checks
- Immutables like Numbers, Tuples, and Strings are hashable, while Lists and Dictionaries are not.
- We use curly brackets `{ }` to create a new set

```
my_set = {“A”, “B”}
print(my_set)
          → {‘B’,‘A’}
my_set.add(“A”)
print(my_set)
          → {‘B’,‘A’}
my_set.add(“C”)
          → {‘B’,‘A’,‘C’}
‘C’ in my_set
          → True
```

# Dictionaries

A `dict` maps a set of keys to arbitrary items.

Like a lookup table: given a **key** value, the `dict` can look up the mapped item. Keys have to be hashable and unique, of course.

```
my_dict = {‘first_name’:‘Al’, ‘last_name’:‘Gebra’}
print(my_dict[‘last_name’])
   → ‘Gebra’
my_dict[last_name] = ‘Igator’
print(my_dict)
   → {‘first_name’:‘Al’, ‘last_name’:‘Igator’}
```

# Other Notable Types

Dates, Times, etc.

# The datetime Module

Standard library’s `datetime` module knows all about dates and times.

```
from datetime import date, datetime
print(date.today())
   → 2017-09-12
print(datetime.now())
   → 2017-09-12 06:18:32.278194
```

# The time Module

The `time` module handles time arithmetic, timezones, formatting, etc.

Fun fact: Time is stored as a number, the number of seconds since midnight of January 1, 1970.

```
import time

time.time()    # the current time

   → 1505125010.978376
```

Fairfield University
DOLAN
School of Business

# Python for Analytics

## Python Data Types

Python Library Docs