Lecture II
Data Types

- Numbers
- Lists & Tuples
- Strings
- Byte Arrays
- Sets
- Dictionaries
- Truth & Nothingness

Numbers

- `int`: Plain integers
- `long`: Arbitrary-length integers.
- `float`: Floating point numbers.
- `complex`: Complex numbers.

Numbers - Integers

- Literals: `789, -100, +912, 0b101, 0o12, 012, 0xAB4, -0x2B, 12L`
- Math, bitwise and comparison operators:
  - Same as C with some extras.
  - ** is the power operator.
    - `7 ** 2 = 49`   `2 ** 10 = 1024`
  - // is the same as /.
- When plain integers exceed size, they are automatically converted to long integers.

Numbers - Floats

- Literals: `0.0`, `5.123`, `6.`, `+1.24`, `-945.2`, `1.2e+78`, `1.2e-78`
- Math and comparison operators:
  - Same as C with some extras.
  - ** is the power operator.
    - `9 ** 1.5 = 27`   `0.5 ** 2 = 0.25`
  - // is "whole number division".
    - `(x // y) == floor(x / y)`
    - `2.0 // 0.5 = 4.0`   `2.0 // 0.55 = 3.0`
- No bitwise operators.
- Limited precision, same as a `double` in C.

Lists & Tuples

- Lists and tuples are both sequence of arbitrary items.
- The only difference is that lists are mutable, while tuples are immutable.
- Both are implemented internally as arrays of pointers.

List & Tuple Literals

- List literals are defined using square brackets:
  - `[]`
  - `[1]`
  - `[1, 2]`
  - `['abc', 4, 'x', [], [2, 'qwe']]`
- Tuple literals are defined using parentheses:
  - `()`
  - `(1,)`
  - `(1, 2)`
  - `('abc', 4, 'x', [], [2, 'qwe'], (5, 1), ())`

Indexing - I

- Lists and tuples are indexed by integers, the same way as C arrays.
  - `x = [6, 7, 8]`
    `x[0]` will return `6`.
    `x[1]` will return `7`.
    `x[2]` will return `8`.
- Indices can be negative, to count in reverse.
  - `x = [6, 7, 8]`
    `x[-1]` will return `8`.
    `x[-2]` will return `7`.
    `x[-3]` will return `6`.

Indexing - II

Slicing

- Portions of lists and tuples can be accessed using "slicing".
- Slicing is taking a part of the list or tuple that consists of several items.
- Slices are defined by *start*, *end*, and optional *step*, separated by colons.
- *Start* and *end* are any valid indices.
- *Step* is an integer specifying the distance between each two consecutive indices.

python

Slicing Example - I

```
>>> x = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']

>>> x[1:3]
['b', 'c']

>>> x[0:3]
['a', 'b', 'c']

>>> x[:3]
['a', 'b', 'c']

>>> x[3:]
['d', 'e', 'f', 'g', 'h', 'i', 'j']

>>> x[2:8]
['c', 'd', 'e', 'f', 'g', 'h']
```

Slicing Example - II

```
>>> x[2:8:2]
['c', 'e', 'g']

>>> x[2:8:1]
['c', 'd', 'e', 'f', 'g', 'h']

>>> x[2:8:3]
['c', 'f']

>>> x[2:8:-2]
[]

>>> x[8:2:-2]
['i', 'g', 'e']

>>> x[::-1]
['j', 'i', 'h', 'g', 'f', 'e', 'd', 'c', 'b', 'a']
```

List & Tuple Operators

- `+` concatenates lists and tuples.
  - `[4, 5, 6] + [1, 2, 3]` → `[4, 5, 6, 1, 2, 3]`
  - `(5, 6) + (3, 5, 0)` → `(5, 6, 3, 5, 0)`
- `*` repeats the list/tuple the specified number of times.
  - `(5, 6) * 3` → `(5, 6, 5, 6, 5, 6)`
  - `[1, 2, 3] * 2` → `[1, 2, 3, 1, 2, 3]`
- `in` checks whether an item is contained in a list/tuple.
  - `3 in (6, 2, 3, 9, 4)` → `True`

List & Tuple Length

- `len(x)` measures the length of the sequence.
  - `len([4, 5, 6])` → `3`
  - `len((5, 6))` → `2`
  - `len((3,))` → `1`
  - `len([5])` → `1`
  - `len(())` → `0`
  - `len([])` → `0`

List & Tuple Methods

- `s.index(x)` returns the first position of `x` in `s`.
  - `(4, 5, 6).index(5)` → 1
  - `(4, 5, 6).index(4)` → 0
  - `(4, 5, 6).index(8)` → ERROR
- `s.count(x)` returns the number of times `x` occurs in `s`.
  - `(4, 5, 6).count(5)` → 1
  - `(4, 5, 5, 2, 5, 7).count(5)` → 3
  - `(4, 2, 6).count(5)` → 0

List Modification

- Unlike tuples, lists can be modified "in-place", i.e. by applying changes to an existing list, instead of creating a new list with the changes.
- List elements and slices can be assigned to.
- Parts of the list can be deleted.
- New items can be inserted into the list.
- The list can be sorted, reversed, etc.

List Item Assignment

- Assigning to individual elements:
  - `x = [1, 2, 3]`
    `x[1] = 8` x ⟶ `[1, 8, 3]`
- Assigning to continuous slices:
  - `x = [1, 2, 3, 4, 5]`
    `x[1:3] = [9, 9, 9, 9]` x ⟶ `[1, 9, 9, 9, 9, 4, 5]`
- Assigning to disjunct slices:
  - `x = [1, 2, 3, 4, 5, 6, 7, 8, 9]`
    `x[1:6:2] = [0, 0, 0]` x ⟶ `[1, 0, 3, 0, 5, 0, 7, 8, 9]`

List Item Removal - I

- The `del` operator can be used to remove single elements and slices:
  - ```
    x = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']
    del x[3]
    x → ['a', 'b', 'c', 'e', 'f', 'g', 'h', 'i']
    del x[2:5]
    x → ['a', 'b', 'g', 'h', 'i']
    ```
    ```

List Item Removal - II

- The `remove` method removes an element given its value (rather than its position):
  - `x = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']`

    `x.remove('f')`

    `x → ['a', 'b', 'c', 'd', 'e', 'g', 'h', 'i']`
- The `pop` method removes an element given its position and returns the removed item:
  - `x = ['a', 'b', 'c', 'd']`

    `y = x.pop(2)`

    `y → 'c'` `x → ['a', 'b', 'd']`

List Item Addition - I

- The `append` method appends an item at the end of a list:
  - ```
    x = ['a', 'b', 'c', 'd']
    x.append(42)
    x → ['a', 'b', 'c', 'd', 42]
    ```
- The `insert` method inserts an item at a particular position in the list:
  - ```
    x = ['a', 'b', 'c', 'd']
    x.insert(2, 42)
    x → ['a', 'b', 42, 'c', 'd']
    ```

List Item Addition - II

- The `extend` method extends the list with the contents of another list:
  - ```
    x = ['a', 'b', 'c', 'd']
    x.extend([2, 5, 6])
    x → ['a', 'b', 'c', 'd', 2, 5, 6]
    ```
    ```

List Sorting

- The `sort` method sorts the list:
  - `x = ['a', 'c', 'd', 'b']`

    `x.sort()`

    `x` → `['a', 'b', 'c', 'd']`
  - `x = ['a', 'c', 'd', 'b']`

    `x.sort(reverse=True)`

    `x` → `['d', 'c', 'b', 'a']`
- Advanced sorting possible, but more complicated.

List Reversion

- The `reverse` method reverses the list:
  - ```
    x = ['a', 'c', 'd', 'b']
    x.reverse()
    x → ['b', 'd', 'c', 'a']
    ```
    ```

Strings

- Strings are sequence of characters or bytes usually used to represent text.
- Ordinary strings are sequences of bytes.
- "Unicode" strings are sequences of characters. Each character may be represented by multiple bytes.
- Unicode strings are useful for non-English text.
- Strings are immutable: all operations on them create new strings.

String Literals - I

- Several ways to define literal strings:
  - Single-line strings: `'abc'`, `"abc"`
  - Multi-line strings:
    - ```
      '''first line
      ...
      last line'''
      ```
    - ```
      """first line
      ...
      last line"""
      ```
- The value of a string does not depend on how the literal is written. This is just for readability.

String Literals - II

- Special characters are represented using the same escape codes as in C.
  - `'first line\nsecond line'`
  - `'first column\tsecond column'`
  - `'\x61\x62\x63'`
  - `'some \'quoted\' text and a slash: \\'`
  - `"more \"quoted\" text."`
- String literals prefixed with an `r` or `R` are "raw" string, which don't interpret escape codes.
  - `r'first line\nstill the same line'`

Unicode String Literals

- Unicode string literals are prefixed with a lowercase or uppercase `u`, and are treated character-by-character rather than byte-by-byte:
  - `x = 'العربية'`

    `y = u'العربية'`

    `x[0]` → '\xd8'   `y[0]` → `u'\u0627' = 'ا'`

Basic String Operations

- Strings are tuples of bytes/characters and behave similarly.
- The addition and multiplication operators are shared.
- The indexing and slicing syntax is the same.

String Functions - I

- Search functions:
  - `find(x)` & `rfind(x)`
  - `index(x)` & `rindex(x)`
  - `count(x)`
  - `startswith(x)` & `endswith(x)`

String Functions - II

- Case conversion functions:
  - `lower()`
  - `upper()`
  - `capitalize()`
  - `title()`
  - `swapcase()`

String Functions - III

- Predicate functions:
  - `islower()`, `isupper()` & `istitle()`
  - `isspace()`
  - `isalpha()`
  - `isdigit()`
  - `isalnum()`

String Functions - IV

- Spacing functions:
  - `lstrip()`, `rstrip()` & `strip()`
  - `ljust()`, `rjust()` & `center()`
  - `zfill()`
  - `expandtabs()`

String Functions - V

- Splitting and joining functions:
  - `split()` & `rsplit()`
  - `partition()` & `rpartition()`
  - `splitlines()`
  - `join(x)`

String Functions - VI

- Replacement function:
  - `replace(x, y)`

String Functions - VII

- Encoding and decoding functions:
  - `encode(x)`
  - `decode(x)`

Byte Arrays

- A `bytearray` is a mutable string.
- Byte arrays support item and slice assignment.
- Byte arrays have all the methods of strings and the following methods of lists:
  - `pop()`
  - `remove(x)`
  - `insert(x, y)`
  - `extend(x)`
  - `append(x)`
- No special literal syntax, so use `bytearray(...)`.

Sets

- A set is an unordered group of unique items.
- Sets are implemented in Python using "hashing".
- Hashing is a technique of storing immutable objects for fast retrieval. It calculates a semi-unique number ("hash") for an object and uses it internally as an array index.
- Hashing does not work on mutable objects because when the object is altered, its hash no longer matches the original.

Sets vs Lists

Lists	Sets
Order Matters	Unordered
Items may repeat	Items are unique
Can store any object	Can store only immutable objects
Slow search	Extremely fast search
Implemented as an array of pointers	Implemented as a hash table

Set Literals

- No special syntax for set literals in Python 2.x. Usually displayed as `set([...])`.
- A set is created by passing a list or tuple to the `set()` constructor:
  - `x = set([1, 2, 3, 2])`

    `x` $\longrightarrow$ `set([1, 2, 3])`
  - `y = set(('abc', (1, 2, 3), 9))`

    `y` $\longrightarrow$ `set(['abc', (1, 2, 3), 9])`

Set Operators

- Sets support the classic mathematical set operators:
  - `&` : intersection.
  - `|` : union.
  - `^` : symmetric difference.
  - `-` : difference.
- Less/More operators compare set size, not contents.
- Equality/Inequality operators compare set contents.

Set Functions - I

- Adding and removing elements:
  - `add(x)`: adds an element.
  - `discard(x)`: removes the element `x` from the set.
  - `remove(x)`: like `discard(x)`, but if `x` is not in the set, raise an error.
  - `pop()`: remove and return an arbitrary element.
  - `clear()`: removes all elements.

Set Functions - II

- Predicates:
  - `isdisjoint(x)`: returns whether two sets share no elements.
  - `issubset(x)`: returns whether `x` is a subset of the set.
  - `issuperset(x)`: returns whether `x` is a superset of the set.

Set Functions - III

- Set operations:
  - `union(x)` & `update(x)`:
    - same as `s | x` & `s |= x` respectively.
  - `intersection(x)` & `intersection_update(x)`:
    - same as `s & x` & `s &= x` respectively.
  - `symmetric_difference(x)` & `symmetric_difference_update(x)`:
    - same as `s ^ x` & `s ^= x` respectively.
  - `difference(x)` & `difference_update(x)`:
    - same as `s - x` & `s -= x` respectively.

Frozen Sets

- Since sets can be modified in place (e.g. by adding new element), they are mutable.
- Since sets are mutable, you can't have sets of sets.
- To solve this, you'll have to use a `frozenset`.
- A `frozenset` is much the same as an ordinary set, but once created, it cannot be altered.
- `frozenset` object do not have element adding/removing methods or any of the four `*update()` methods.

Dictionaries

- A dictionary is a mapping from a set of keys to a group of values.
- Also called "associative arrays", "maps" or "hash tables" in other languages.
- Each key, value pair is called an "item".
- Implemented the same way as sets, except for each set item, there is a related object of arbitrary type.
- Notable for efficiency and flexibility.
- Keys must be immutable objects, while values can be anything.

Dictionary Literals

- Dictionaries are defined using braces, items are separated by commas, each key and value are separated a colon:
  - `{'calculus': 78, "arabic": 63, 'C': 80, 'C++': 91}`
  - ```
    {42: 'the answer',
     'hello': 'world',
     (9, 8, 7): '!',
     (1, 'a'): ['abc', 1.23],
     3.15169: 'pi'}
    ```
- Can also be constructed by calling `dict`:
  - `dict(calculus=78, arabic=53, C=96)`
    ```

Dictionary Access

- Dictionaries are indexed with square brackets, the same way as sequence types:
  - x = {'calculus': 78, "arabic": 63, 'C': 80, 'C++': 91}

    x['C++'] → 91

    x['arabic'] → 63

    x['statistics'] → ERROR
- Slicing does not make sense for dictionaries, as values are unordered, so it is not supported.

Dictionary Modification

- The values of dictionary items are added and modified by assigning to an index:
  - ```
    x = {'a': 1, 'b': 2, 'c': 3}
    x['a'] = 50
    x → {'a': 50, 'b': 2, 'c': 3}
    x['x'] = 'hello'
    x → {'a': 50, 'b': 2, 'c': 3, 'x': 'hello'}
    ```
- Items can be deleted using the `del` operator:
  - ```
    del x['b']
    x → {'a': 50, 'c': 3, 'x': 'hello'}
    ```

Dictionary Functions - I

- The `has_key(x)` method checks whether a key exists in the dictionary:
  - `x = {'a': 9, 'b': 8, 'c': 'q'}`

    ```
    x.has_key('a') → True
    x.has_key('t') → False
    x.has_key('q') → False
    ```
- The `in` operator works identically to `has_key(x)`:
  - `x = {'a': 9, 'b': 8, 'c': 'q'}`

    ```
    'a' in x → True
    't' in x → False
    'q' in x → False
    ```

Dictionary Functions - II

- The `pop(x)` method removes an item given its key and returns its value:
  - ```
    x = {'a': 9, 'b': 8, 'c': 'q'}
    y = x.pop('a')
    y → 9           x → {'b': 8, 'c': 'q'}
    ```
- The `popitem()` method removes and returns an arbitrary item:
  - ```
    x = {'a': 9, 'b': 8, 'c': 'q'}
    y = x.popitem()
    y → ('b', 8)   x → {'a': 9, 'c': 'q'}
    ```

Dictionary Functions - III

- The `clear()` method removes all items from the dictionary:
  - ```
    x = {'a': 9, 'b': 8, 'c': 'q'}
    x.clear()
    x → {}
    ```
- The `update(x)` method merges a new dictionary into an existing one:
  - ```
    x = {'a': 9, 'b': 8, 'c': 'q'}
    y = {'m': 6, 'b': 1}
    x.update(y)
    x → {'a': 9, 'b': 1, 'm': 6, 'c': 'q'}
    ```

Dictionary Functions - IV

- The `keys()`, `values()` and `items()` methods each return a list of the dictionary's keys, values or items respectively in arbitrary order:
  - `x = {'a': 9, 'b': 8, 'c': 'q'}`

    ```
    x.keys() → ['c', 'a', 'b']
    x.values() → [9, 'q', 8]
    x.items() → [('b', 8), ('a', 9), ('c', 'q')]
    ```
- The `iterkeys()`, `itervalues()` and `iteritems()` methods are similar to the above but return iterators rather than lists (more about iterators later).
- All the above methods are useful in `for` loops.

Truth & Nothingness

- The built-in symbol `None` is used to represent nothingness, or the lack of value. It is similar to "null" in other languages.
- Python has a `bool` type to represent Boolean values.
- Boolean objects take of of two values, `True` and `False`.
- When used in a Boolean context (e.g. as a condition), non-Boolean values are converted to Boolean ones.

Truth of Non-Booleans

- The following values are `False` in Boolean contexts:
  - `None`
  - 0 of any numeric type.
  - Any object `x` for which `len(x) = 0`. These include:
    - Empty sequences: `[]`, `()`, `""`, `bytearray('')`.
    - Empty sets: `set([])`, `frozenset([])`.
    - Empty dictionaries: `{}`.
    - Instances of classes that define length whose length is zero.

Boolean Operations

- The three well-known Boolean operations are carried out in Python using the operators `and`, `or` and `not`.
  - `True and False` ⟶ `False`
  - `(True or False) and True` ⟶ `True`
  - `not True or not False` ⟶ `True`
- The `and` and `or` operators are both "short-circuited". They don't evaluate the second operand unless necessary:
  - `f() and g()` will not call `g()` if `f()` is `g()` if `False`.
  - `f() or g()` will not call `g()` if `f()` is `g()` if `True`.