Objectives

- Be able to specify the uses and effects of the Python operators and to use them in natural ways, taking into account their precedences
- Be able to describe what is meant by a keyed collection and to say what data structures in Python are keyed collections
- Be able to specify the uses and effects of the built-in Python data structures (collections) and to use them in natural ways
- Be able to use Python subscripting and slicing
- Be able to take advantage of Python tuple assignment
- Be able to classify Python data types as mutable or immutable, and to demonstrate what is meant by those terms

Some Object-Oriented Terminology

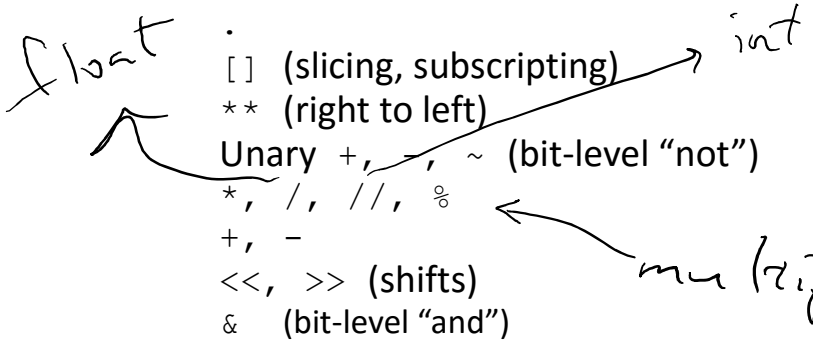
- C++/Python camp:
  - *Member functions* are *called* to invoke *behaviors* of *objects*

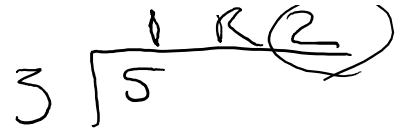
- Smalltalk / Objective C / Swift camp:
  - *Methods* are used to send *messages* to *objects,* again invoking *behaviors*

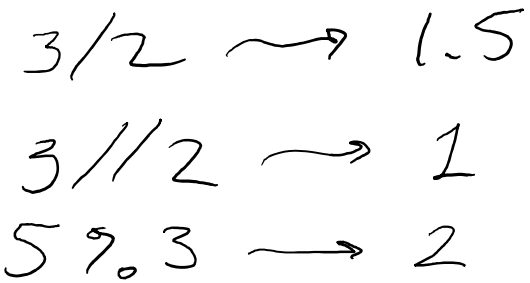
Operator Precedence

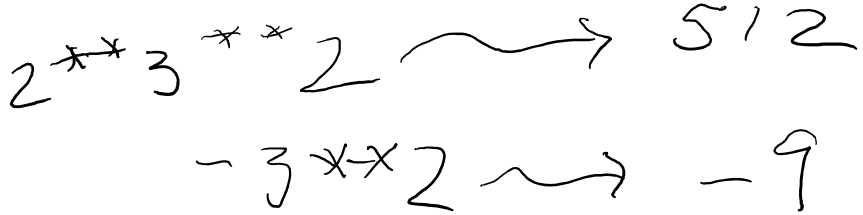
- The following list is in order from most tightly to least tightly binding. Function call precedence is not listed, but can safely be considered the same as [] (slicing and subscripting):

```
.
[] (slicing, subscripting)
** (right to left)
Unary +, -, ~ (bit-level "not")
*, /, //, %
+, -
<<, >> (shifts)
&   (bit-level "and")
^   (bit-level "exclusive or")
|   (bit-level "or")
is, is not, ==, !=, <, >, <=, >=
not
and
or
```









Python collections

- All Python collections are heterogeneous (NOT required to be all same type) except str, byte, and bytearray

Keyed Collections

- set and dict are *keyed*, not subscripted in the ordinary sense…

Length function

- All built-in data collections in Python support the *len* function

All non-keyed Python data structures support

- Subscripting

- Slices

Lists

- *Lists* (type `list`) are mutable, variable-sized, sequential, heterogeneous collections
- List construction syntax is provided by the language, and consists of a comma-separated list of expressions delimited by square brackets.
- Example: `[89, 3.45,'ham',[7,8]]`
- Can grow the list from the rear with the `append()` message, and remove items with the `pop()` message or the Python `del` primitive.

More about Lists

- Lists are mutable:

```
>>> a = [21, 7j, 'Abe']
>>> a[1] += 1
>>> a.append('Mary')
>>> a
[21, (1+7j), 'Abe', 'Mary']
```

- Slices can receive values:

```
>>> a[0:2] = ["Robert"]
>>> a
['Robert', 'Abe', 'Mary']
```

Some other member functions of class list

- append
- pop
- sort
- reverse
- index
- count
- remove

`str` is *immutable*

- Can delimit a string literal with single, double, or triple quotes
- Example:

```
mandy = 'Mindy'
mandy[1] = 'a' #Illegal. Can't change a string.
```

- Instead, do…

```
mandy = mandy[0]+'a'+mandy[2:]
```

- There is no "single-character" type. Python uses a one-character string instead. If s is a one-character string, then `ord(s)` is the character code of that one character.
- If `m` is a small positive integer, then `chr(m)` is a one-character string whose only element has character code `m`.

Type `bytes`

- Objects of type `str` are strings of 16-bit characters.
- Objects of type `bytes` are strings of 8-bit characters.
- Like `str`, `bytes` is homogeneous and immutable.
- A `bytes` literal is like a `string` literal, but prefixed with the character `b`.
  - The characters in `b'Chicken'` occupy seven bytes.
  - The characters in `'Chicken'` occupy fourteen bytes.

Type `bytearray`

- Closely related to `bytes` is the other mutable sequence, `bytearray`.
- This type has no literals, but a `bytes` object can be used to construct a `bytearray` object, as follows…

```
>>> a = bytearray(b'The week is over')
>>> a[0:8] = b'It'
>>> a
bytearray(b'It is over')
```

Input and Output

Type `tuple`

- `tuple` is an immutable, heterogeneous sequence type.
- A comma-separated list of expressions, enclosed in parentheses, is a tuple-valued expression
- Note that although a tuple is immutable, if one of its elements is mutable that element can be sent value-changing messages.
  - It is the *identity* of the object that cannot be changed

More `tuple` particulars

- When there is no danger of ambiguity, the parentheses enclosing a tuple expression can be omitted.

```
>>> x = 21, [34, 12], 'Ice cream'
>>> x
(21, [34, 12], 'Ice cream')
```

Tuple assignment

- *Tuple assignment* allows the individual elements of a sequence to be assigned to corresponding variables on the left-hand-side of the assignment operator.
- Examples:

```
(dad, mom) = ("George", "Martha")
(pi, e) = 3.14159, 2.71828
vegetable, meat, drink, dessert = \
    [ 'carrots', 'steak', 'tea', 'pie' ]
```

Returning a tuple from a function

- This interpretation of assignment provides an interesting way for a function to return more than one value. As an example, consider the following function.

```
def divide(dividend, divisor):
    return dividend//divisor, dividend%divisor
```

- This function returns a two-element tuple and could be called as follows.

```
quo, rem = divide(25, 7)
```

More String Operations

- The % operator relies on format codes similar to those used in the C language `printf()` library function.
- A similar (seemingly redundant at first glance) facility also present in Python employs ordinal numbers in curly braces and a `format()` message.

```
labeling = \
        "{0} cares nothing for {1}; she only cares for {0}."
print(labeling.format("Emma", "Paul"))
```

Keys

- Among built-in types, only immutable types can be keys for a set or dict.
- This is because changing an object's value once it has been placed in one of these keyed data structures can make that object unreachable
- Programmer-defined types may violate this principle, because Python does not attempt to check new types to ensure they are immutable – that is the programmer's responsibility

Type `set`

- The following code creates a keys-only collection called a *set* and places three values in it.

```
s = set()
s.add(39)
s.add('Lois')
s.add(5+1j)
```

- Or we could do the same thing using a set-valued expression, as follows:

```
s = {39, 'Lois', 5+1j}
```

Type `dict`

- Objects of type `dict` store (key, value) pairs.
- The following code creates a dict object.

```
b = dict()
b['Ralph'] = 1949
b['Jen'] = 1980
```

- Equivalent is…

```
b = {'Ralph':1949, 'Jen':1980}
```

- Or…

```
b = dict(Ralph=1949, Jen=1980)
```

Control Structures – *if, if…else,* and *if…elif…else*

Control Structures - *while*

Exception Handling

Defining functions

Defining classes

Scopes – block scope, class scope, and object scope