CSc 120
Introduction to Computer Programming II

Adapted from slides by Dr. Saumya Debray

01: Python review
this class in context
Computer programming
Computer programming

- Python code (CSc 120)
- Java code (CSc 210)
- C code (CSc 352)

- Python interpreter
- Java compiler + JVM
- C compiler (CSc 453)

operating system (CSc 452)

computer
Computer programming

this class

Python code (CSc 120)

Java code (CSc 210)

C code (CSc 352)

Python interpreter

Java compiler + JVM

C compiler (CSc 453)

operating system (CSc 452)

programmer

computer
getting started
Python language and environment

- Language: Python 3
  - free download from https://www.python.org/downloads
  - documentation available at https://www.python.org/doc
    - tutorial
    - beginner's guide
    - language reference
    - setup and usage, HOWTOs, FAQs
Python language and environment

- Programming environment: idle (or idle3)
  - comes bundled with Python download
  - provides:
    - interactive Python shell
    - debugger
    - execution from a file
Surprises if coming from C, C++, Java

• No variable declarations
• Indentation instead of { }
• Flexible for loop
• Built-in data structures (lists, dictionaries, tuples, sets)
• Arbitrary-precision integers
• Devisison differences
• Garbage collection (also in Java)
  o no explicit allocation/deallocation
python review: variables, expressions, assignment
python basics

```python
>>> x = 4
>>> y = 5
>>> z = x + y
>>> x
4
>>> y
5
>>> z
9
>>> y = z * 2
>>> y
18
>>> 
```
python basics

>>> x = 4
>>> y = 5
>>> z = x + y
>>> x
4
>>> y
5
>>> z
9
>>> y = z * 2
>>> y
18

>>> : python interpreter's prompt
black: user input (keyboard)
blue: python interpreter output
python basics

>>> x = 4
>>> y = 5
>>> z = x + y
>>> x
4
>>> y
5
>>> z
9
>>> y = z * 2
>>> y
18
>>>
python basics

```python
>>> x = 4
>>> y = 5
>>> z = x + y
```

```
4
>>> y
5
>>> z
9
>>> y = z * 2
```

```
18
>>> y
expressions
```
python basics

```python
>>> x = 4
>>> y = 5
>>> z = x + y
>>> x
4
>>> y
5
>>> z
9
>>> y = z * 2
>>> y
18
>>> 
```
typing in an expression causes its value to be printed
python basics

>>> x = 4
>>> y = 5
>>> z = x + y
>>> x
4
>>> y
5
>>> z
9
>>> y = z * 2
>>> y
18

• variables:
  - names begin with letter or ' '_
  - don't have to be declared in advance
    - type determined at runtime

• expressions:
  - all the usual arithmetic operators
Multiple (aka parallel) assignment

```python
>>> x, y, z = 11, 22, 33
>>> x
11
>>> y
22
>>> z
33
>>> Assigns to multiple variables at the same time
```

$x_1, x_2, ..., x_n = \exp_1, \exp_2, ..., \exp_n$

Behavior:

1. $\exp_1, ..., \exp_n$ evaluated (L-to-R)
2. $x_1, ..., x_n$ are assigned (L-to-R)
EXERCISE

```python
>>> x = 3
>>> y = 4
>>> z = (2*x - 1 == y+1)
```

`what value is printed out for z?`
EXERCISE

>>> x = 3
>>> y = 4
>>> sum, diff, prod = x + y, x - y, x * y
>>> prod + diff

← what is the value printed out?
python review: reading user input

```python
input()
```
>>> x = input()
13579

>>> x
'13579'

>>> y = input('Type some input: ')
Type some input: 23

>>> y
'23'

>>> z = input('More input: ')
More input: 567

>>> z
'567'

>>>
Reading user input I: input()

```python
>>> x = input()
13579
>>> x
'13579'
>>> y = input('Type some input: ')
Type some input: 23
>>> y
'23'
>>> z = input('More input: ')
More input: 567
>>> z
'567'
```
Reading user input I: `input()`

```python
>>> x = input()
13579
>>> x
'13579'
```

Input statement:
- reads input from the keyboard
- returns the value read
  - (a string)
- takes an optional argument
  - if provided, serves as a prompt

```python
>>> y = input('Type some input: ')
Type some input: 23
>>> y
'23'

>>> z = input('More input: ')
More input: 567
>>> z
'567'
```

```python

```
Reading user input I: \texttt{input()}

```python
>>> x = \texttt{input()}
12
>>> x
'12'
>>> y = x / 2
Traceback (most recent call last):
  File "<pyshell#59>", line 1, in <module>
    y = x / 2
TypeError: unsupported operand type(s) for /: 'str' and 'int'
>>> 
```

the value read in is represented as a string
• string $\equiv$ sequence of characters
Reading user input I: `input()`

```python
>>> x = input()
12
>>> x
'12'
>>> y = x / 2
Traceback (most recent call last):
  File "<pyshell#59>", line 1, in <module>
    y = x / 2
TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

The value read in is represented as a string
- string ≡ sequence of characters

- `TypeError`: indicate an error due to wrong type
the value read in is represented as a string
• string \(\equiv\) sequence of characters
• TypeError: indicates an error due to a wrong type

• Fix: explicit type conversion
python review: basics of strings
Basics of strings

```python
>>> x = "abcd"
>>> y = 'efgh'
>>> z = "efgh"
>>> ```
Basics of strings

```python
>>> x = "abcd"
>>> y = 'efgh'
>>> z = "efgh"
```
Basics of strings

```python
>>> text = input('Enter a string: ')
Enter a string: abcdefghi
>>> text
'abcdefghi'
>>> text[0]
'a'
>>> text[1]
'b'
>>> text[27]
Traceback (most recent call last):
  File "<pyshell#153>", line 1, in <module>
    text[27]
IndexError: string index out of range
```

A string is a sequence (array) of characters.
- We can index into a string to get the characters.

---

31
Basics of strings

>>> text = input('Enter a string: ')
Enter a string: abcdefghi
>>> text
'abcdefghi'
>>> text[0]
'a'
>>> text[1]
'b'
>>> text[27]
Traceback (most recent call last):
  File "<pyshell#153>", line 1, in <module>
    text[27]
IndexError: string index out of range

a string is a sequence (array) of characters
• we can index into a string to get the characters

indexing beyond the end of the string gives an IndexError error
Basics of strings

```python
>>> text = input('Enter a string: ')
Enter a string: abcdefghi
>>> text
'abcdefghi'
>>> text[0]
'a'
>>> text[1]
'b'
>>> text[27]
Traceback (most recent call last):
  File "<pyshell#153>", line 1, in <module>
text[27]
IndexError: string index out of range
```
Basics of strings

```python
>>> x = '0123456789'
>>> x[0]
'0'
>>> x[1]
'1'
>>> x[2]
'2'
>>> x[-1]
'9'
>>> x[-2]
'8'
>>> x[-3]
'7'
```

\[ x[i] : \text{if } i \geq 0 \text{ (i.e., non-negative values):} \]
\[ \begin{array}{l}
\text{• indexing is done from the beginning of the string} \\
\text{• the first letter has index 0}
\end{array} \]

\[ x[i] : \text{if } i < 0 \text{ (i.e., negative values):} \]
\[ \begin{array}{l}
\text{• indexing is done from the end of the string} \\
\text{• the last letter has index -1}
\end{array} \]
Basics of strings

```python
>>> x = '0123456789'
>>> x[0]
'0'
>>> x[1]
'1'
>>> x[2]
'2'
>>> x[-1]
'9'
>>> x[-2]
'8'
>>> x[-3]
'7'
```

- $x[i]$: if $i \geq 0$ (i.e., non-negative values):
  - indexing is done from the beginning of the string
  - the first letter has index 0

- $x[i]$: if $i < 0$ (i.e., negative values):
  - indexing is done from the end of the string
  - the last letter has index -1
EXERCISE

```python
>>> x = 'a'
>>> x == x[0]
```

what do you think will be printed here?
EXERCISE

```python
>>> x = 'apple'
>>> x[2] == x[-2]
```

What do you think will be printed here?
Basics of strings

Inside a computer, a character is represented as a number (its "ASCII value")

```python
>>> x = '5'
>>> x
'5'
>>> x == 5
False
```
Basics of strings

Inside a computer, a character is represented as a number (its "ASCII value")

the ASCII value of a digit is not the same as the digit itself:

'5' ≠ 5
EXERCISE

```python
>>> x = 27
>>> y = 'x'
>>> x == y
```

What do you think will be printed here? Why?
Basics of strings

```python
>>> x = input()
abcDE_fgHIJ_01234
>>> x
'abcDE_fgHIJ_01234'
>>> len(x)
17
>>> y = x.lower()
>>> y
'abcde_fghij_01234'
>>> x = y.upper()
>>> x
'ABCDEF_GHIJ_01234'
```

len(x) : length of a string x
Basics of strings

```python
>>> x = input()
abcDEF_gHIJ_01234
>>> x
'abcDEF_gHIJ_01234'
>>> len(x)
17
>>> y = x.lower()
>>> y
'abcde_fghij_01234'
>>> len(y)
17
>>> x = y.upper()
>>> x
'ABCDE_FGHIJ_01234'
```

- `len(x)`: length of a string `x`
- `x.lower()`, `x.upper()`: case conversion on the letters in a string `x`
  - note that non-letter characters are not affected
Basics of strings

```python
>>> x = input()
abcDE fgHIJ 01234

>>> x
'abcDE fgHIJ 01234'

>>> x

>>> len(x)
17

>>> y = x.lower()

>>> y
'abcde_fghij_01234'

>>> len(x) : length of a string x

x.lower(), x.upper() : case conversion on the letters in a string x
• note that non-letter characters are not affected

Python supports a wide variety of string operations
• see www.tutorialspoint.com/python3/python_strings.htm

>>> x = y.upper()

>>> x
'ABCDE FGHIJ 01234'

>>> `
Basics of strings

```python
>>> x = input()
abcdefgh

>>> x
'abcdefgh'

>>> x[3]
'd'

>>> x[3] = 'z'
Traceback (most recent call last):
  File "<pyshell#193>", line 1, in <module>
    x[3] = 'z'
TypeError: 'str' object does not support item assignment
```
Basics of strings

>>> x = input()
abcdefgh

strings are immutable, i.e., cannot be modified or updated

>>> x
'abcdefgh'

>>> x[3]
'd'

>>> x[3] = 'z'
Traceback (most recent call last):
  File "<pyshell#193>", line 1, in <module>
    x[3] = 'z'
TypeError: 'str' object does not support item assignment

>>>
Basics of strings

Strings are immutable, i.e., cannot be modified or updated. To "modify" a string, we have to create a copy of it with the appropriate part(s) replaced by the new values.
Basics of strings

Strings are immutable, i.e., cannot be modified or updated. To "modify" a string, we have to create a copy of it with the appropriate part(s) replaced by the new values. These operations are called "slicing."
Basics of strings

Strings are **immutable**, i.e., cannot be modified or updated. To "modify" a string, we have to create a copy of it with the appropriate part(s) replaced by the new values. These operations are called "slicing" + applied to strings does concatenation.

```python
>>> x = input()
abcdefg

>>> x
'abcdefg'

>>> x[3]
'd'

>>> x[3] = 'e'
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    x[3] = 'e'
TypeError: 'str' object does not support item assignment

>>> x[:3] + 'e' + x[4:]
'abcdefg'

>>> (x[3] + 'xyz' + x[5:7])
'abcxyzfg'
```
Basics of strings

+ applied to strings does concatenation
Basics of strings

+ applied to strings does concatenation

'*' applied to strings:
• does repeated concatenation if one argument is a number
• generates an error otherwise
Basics of strings

+ applied to strings does concatenation

* applied to strings:
  - does repeated concatenation if one argument is a number
  - generates an error otherwise

not all arithmetic operators carry over to strings
EXERCISE

```python
>>> x = "whoa!"
>>> y = x[2] * len(x)
>>> z = x[3] + x[0] + y
>>> z
awoooooo
```

what is printed here?
EXERCISE

```python
>>> x = input()
>>> y = x + x
>>> int(x) == int(y)
True
```

What input value(s) will cause this to work as shown?
python review: conditionals
Conditional statements: if/elif/else

```python
>>> var1 = input()
100
>>> var2 = input()
200
>>> x1 = int(var1)
>>> x2 = int(var2)

>>> if x1 > x2:
    print('x1 is bigger than x2')
elif x1 == x2:
    print('x1 and x2 are equal')
else:
    print('x1 is smaller than x2')

x1 is smaller than x2
```
Conditional statements: if/elif/else

```python
>>> var1 = input()
100
>>> var2 = input()
200
>>> x1 = int(var1)
>>> x2 = int(var2)
>>> if x1 > x2:
    print('x1 is bigger than x2')
elif x1 == x2:
    print('x1 and x2 are equal')
else:
    print('x1 is smaller than x2')
x1 is smaller than x2
```
Conditional statements: if/elif/else

```python
>>> var1 = input()
100
>>> var2 = input()
200
>>> x1 = int(var1)
>>> x2 = int(var2)

# If-statement syntax:
if BooleanExpr:
  stmt

elif BooleanExpr:
  stmt

elif ...
...

else:
  stmt

# Elif's are optional (use as needed)
else is optional

# Example usage:
>>> if x1 > x2:
    print('x1 is bigger than x2')
elif x1 == x2:
    print('x1 and x2 are equal')
elif ...
...
else:
    stmt
...

x1 is smaller than x2
```
python review: while loops
Loops I: while

```python
Python 3.4.3 (default, Sep 14 2016, 12:36:27)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.
>>> N = input('N: ')
N: 5
>>> limit = int(N)
>>> i = 0
>>> sum = 0
>>> while i <= limit:
    sum += i
    i += 1

>>> sum
15
>>> 
```
Loops I: while

- **while**-statement syntax:

```
while BooleanExpr :
  stmt_1
  ...
  stmt_n
```

- $stmt_1 \ldots stmt_n$ are executed repeatedly as long as $BooleanExpr$ is True
python review: 
lists (aka arrays)
Lists

```
Python 3.4.3 (default, Sep 14 2016, 12:36:27)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.
>>> x = [ 'item1', 'item2', 'item3', 'item4' ]
>>> x[0]
'item1'
>>> x[2]
'item3'
>>> len(x)
4
>>> x[2] = 'newitem3'
>>> x
['item1', 'item2', 'newitem3', 'item4']
>>> x[1:]
['item3', 'newitem3', 'item4']
>>> x[:3]
['item1', 'item2', 'newitem3']
>>> x[1:3]
['item2', 'newitem3']
>>> |
```
Lists

A list (or array) is a sequence of values.
Lists

A list (or array) is a sequence of values accessing list elements (i.e., indexing), computing length: similar to strings

- non-negative index values ($\geq 0$) index from the front of the list
  - the first element has index 0
- negative index values index from the end of the list
  - the last element has index -1
EXERCISE

```python
>>> x = [ "abc", "def", "ghi", "jkl" ]
>>> x[1] + x[-1]
```

What do you think will be printed here?
Lists

A list (or array) is a sequence of values. Accessing list elements (i.e., indexing), computing length: similar to strings. Lists are **mutable**, i.e., can be modified or updated.

- different from strings
Lists

A list (or array) is a sequence of values accessing list elements (i.e., indexing), computing length: similar to strings lists are *mutable*, i.e., can be modified or updated
• different from strings

slicing: similar to strings
Lists

concatenation (+ and *) : similar to strings
Lists

concatenation (+ and *) : similar to strings

these operators create “shallow” copies

• due to list mutability, this can cause unexpected behavior
Lists

concatenation (+ and *) : similar to strings

these operators create “shallow” copies
  • due to list mutability, this can cause unexpected behavior
Lists

concatenation (+ and *) : similar to strings
these operators create “shallow” copies
• due to list mutability, this can cause unexpected behavior

Python 3.4.3 (default, Nov 17 2016, 01:08:31)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.
>>> x = [ [12, 34, 56] ]
>>> 
>>> y = x * 3
>>> y
[[[12, 34, 56], [12, 34, 56], [12, 34, 56]]
>>> 
>>> y[0].append(78)
>>> 
>>> y
[[[12, 34, 56], [12, 34, 56], [12, 34, 56]], [12, 34, 56, 78]]
>>> 

x

y

12 34 56

12 34 56

12 34 56 78

after y[0].append(78)
Lists: sorting

```
Python 3.4.3 (default, Sep 14 2016, 12:36:27)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.

>>> x = [1,4,3,2,5]
>>> x
[1, 4, 3, 2, 5]
>>> x.sort()
>>> x
[1, 2, 3, 4, 5]
>>> y = [1,4,3,2,5]
>>> y
[1, 4, 3, 2, 5]
>>> sorted(y)
[1, 2, 3, 4, 5]
>>> y
[1, 4, 3, 2, 5]
>>> sorted(y, reverse=True)
[5, 4, 3, 2, 1]
>>> y
[1, 4, 3, 2, 5]
```
Lists: sorting

sort() : sorts a list

sorted() : creates a sorted copy of a list; the original list is not changed
Lists: sorting

sort() : sorts a list

sorted() : creates a sorted copy of a list; the original list is not changed

the optional argument reverse specifies ascending or descending order
python review:
lists ↔ strings
Strings → lists

split() : splits a string on whitespace, returns a list of strings
Strings → lists

split() : splits a string on whitespace returns a list of strings

split(delim) : given an optional argument
delim, splits the string on delim
Lists → strings

delim.join(list) : joins the strings in list using the string delim as the delimiter
returns a string
String trimming

what if we wanted to get rid of the punctuation?
String trimming

```
x.strip() : removes whitespace from either end of the string x
returns a string
```
String trimming

x.strip() : removes whitespace from either end of the string x

x.strip(string) : given an optional argument string, removes any character in string from either end of x
String trimming

```python
>>> x = " abc "
>>> x
' abc '
>>> x.strip()
'abc'
>>> y = "Hey!!!"
>>> y
'Hey!!!'
>>> y.strip("!")
'Hey'
>>> z = "!#$%stuff stuff stuff ^&*()_+
>>> z
'!#$%stuff stuff stuff ^&*()_+
>>> z.strip("!#$%&*()_+
'stuff stuff stuff'
```
what if we wanted to get rid of the punctuation?

- iterate over the list
- use strip() to trim each word in the list
- reassemble the trimmed words into a list
String trimming

what if we wanted to get rid of the punctuation?

- iterate over the list
- use strip() to trim each word in the list
- reassemble the trimmed words into a list
python review: functions
Functions

- **def fn_name ( arg₁, ..., argₙ )**
- defines a function `fn_name` with n arguments `arg₁, ..., argₙ`
Functions

• **def fn_name ( arg₁, ..., argₙ )**
  - defines a function *fn_name* with *n* arguments *arg₁, ..., argₙ*

• **return expr**
  - optional
  - returns the value of the expression *expr* to the caller
Functions

- **def fn_name ( arg₁, ..., argₙ )**
  - defines a function *fn_name* with *n* arguments *arg₁, ..., argₙ*

- **return expr**
  - optional
  - returns the value of the expression *expr* to the caller

- **fn_name(expr₁, ..., exprₙ)**
  - calls *fn_name* with arguments *expr₁, ..., exprₙ*
python review: reading user input II: file I/O
suppose we want to read (and process) a file "this_file.txt"
Reading user input II: file I/O

- open() the file
- read and process the file
- close() the file
Reading user input II: file I/O

- `fileobj = open(filename)`
- `filename`: a string
- `fileobj`: a file object
Reading user input II: file I/O

- `fileobj = open(filename)`
  - `filename`: a string
  - `fileobj`: a file object

- `for var in fileobj`:
  - reads the file a line at a time
  - assigns the line (a string) to `var`
Reading user input II: file I/O

- `fileobj = open(filename)`
  - `filename`: a string
  - `fileobj`: a file object
- `for var in fileobj:`
  - reads the file a line at a time
  - assigns the line (a string) to `var`
- Note that each line read ends in a newline (`\n`) character
Reading user input II: file I/O

at this point we've reached the end of the file so there's nothing left to read
Reading user input II: file I/O

At this point we've reached the end of the file so there's nothing left to read.

To re-read the file, we have to close it and then re-open it.
at this point we've reached the end of the file so there's nothing left to read
to re-read the file, we have to close it and then re-open it

NOTE: we can use strip() to get rid of the newline character at the end of each line
Writing output to a file

```python
open(filename, "w") : opens filename in write mode, i.e., for output
```
Writing output to a file

open(filename, "w") : opens filename in write mode, i.e., for output

fileobj.write(string) : writes string to fileobj
Writing output to a file

=open(filename, "w") : opens filename in write mode, i.e., for output

=fileobj.write(string) : writes string to fileobj

open the file in read mode ("r") to see what was written
python review:
tuples
Tuples

A tuple is a sequence of values (like lists)
Tuples

A tuple is a sequence of values (like lists)

- Tuples use parens ()
  - By contrast, lists use square brackets []
    - Parens can be omitted if no confusion is possible

- Special cases for tuples:
  - Empty tuple: ()
  - Single-element tuple: must have comma after the element:
    (111,)
Tuples

A tuple is a sequence of values (like lists)
tuples use parens ()
• by contrast, lists use square brackets []
  • parens can be omitted if no confusion is possible
• special cases for tuples:
  • empty tuple: ()
  • single-element tuple: must have comma after the element:

(111,)

Indexing in tuples works similarly to strings and lists
Tuples

computing a length of a tuple: similar to strings and lists
Tuples

computing a length of a tuple: similar to strings and lists

computing slices of a tuple: similar to strings and lists
Tuples

+ and * work similarly on tuples as for lists and strings
Tuples

Iterating through the elements of a tuple: similar to lists and strings

```python
>>> x = (111, 222, 333, 444, 555)
>>> for y in x:
    print(y)
111
222
333
444
555
>>> 222 in x
True
>>> 999 in x
False
```
Tuples

- Iterating through the elements of a tuple: similar to lists and strings
- Checking membership in a tuple: similar to lists and strings
Tuples

Tuples are not mutable.

```python
>>> x = (111, 222, 333, 444, 555)
>>> x[2]
133
>>> x[2] = 999
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    x[2] = 999
TypeError: 'tuple' object does not support item assignment
```
Sequence types: mutability

tuples are immutable
Sequence types: mutability

tuples are immutable

lists are mutable (even if the list is an element of a [immutable] tuple)
Sequence types: mutability

tuples are immutable

lists are mutable (even if the list is an element of a [immutable] tuple)

strings are immutable (even if the string is an element of a [mutable] list)
Sequence types: mutability

```
>>> x = (['aaa', 'bbb'], ['ccc', 'ddd'], ['eee'])
>>> x[0] = 'fff'
Traceback (most recent call last):
  File "<ipython-input-3-55352f420e1e>", line 1, in <module>
    x[0] = 'fff'
TypeError: 'tuple' object does not support item assignment
>>> x[0][0] = 'fff'
>>> x
(['fff', 'bbb'], ['ccc', 'ddd'], ['eee'])
>>> x[0][0] = 'a'
Traceback (most recent call last):
  File "<ipython-input-3-55352f420e1e>", line 1, in <module>
    x[0][0] = 'a'
TypeError: 'str' object does not support item assignment
```
Sequence types: mutability

tuple (immutable)

list (mutable)

string (immutable)
Why use tuples?

At the implementation level, tuples are much simpler than lists:

• lists are mutable; tuples are immutable
  • this means that the implementation can process tuples without having to worry about the possibility of updates

• lists have methods (e.g., append); tuples do not have methods

⇒ Tuples can be implemented more efficiently than lists
Sequence types include: strings, lists, and tuples

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x in s</code></td>
<td>True if an item of <code>s</code> is equal to <code>x</code>, else False</td>
</tr>
<tr>
<td><code>x not in s</code></td>
<td>False if an item of <code>s</code> is equal to <code>x</code>, else True</td>
</tr>
<tr>
<td><code>s + t</code></td>
<td>the concatenation of <code>s</code> and <code>t</code></td>
</tr>
<tr>
<td><code>s * n</code> or <code>n * s</code></td>
<td>equivalent to adding <code>s</code> to itself <code>n</code> times</td>
</tr>
<tr>
<td><code>s[i]</code></td>
<td><code>i</code>th item of <code>s</code>, origin 0</td>
</tr>
<tr>
<td><code>s[i:j]</code></td>
<td>slice of <code>s</code> from <code>i</code> to <code>j</code></td>
</tr>
<tr>
<td><code>s[i:j:k]</code></td>
<td>slice of <code>s</code> from <code>i</code> to <code>j</code> with step <code>k</code></td>
</tr>
<tr>
<td><code>len(s)</code></td>
<td>length of <code>s</code></td>
</tr>
<tr>
<td><code>min(s)</code></td>
<td>smallest item of <code>s</code></td>
</tr>
<tr>
<td><code>max(s)</code></td>
<td>largest item of <code>s</code></td>
</tr>
<tr>
<td><code>s.index(x[, i[, j]])</code></td>
<td>index of the first occurrence of <code>x</code> in <code>s</code> (at or after index <code>i</code> and before index <code>j</code>)</td>
</tr>
<tr>
<td><code>s.count(x)</code></td>
<td>total number of occurrences of <code>x</code> in <code>s</code></td>
</tr>
</tbody>
</table>

The elements are: \(i, i+k, i+2k, \ldots\)

Source: https://docs.python.org/3/library/stdtypes.html#sequence-types-list-tuple-range
EXERCISE

```python
>>> x = [ (1, 2, 3), (4, 5, 6), (7, 8, 9) ]
```

```python
>>> x[0][0] = (2, 3, 4)
```

```
>>> x[0] = [ 2, 3, 4 ]
```

what do you think will be printed out?

what do you think will be printed out?
python review: dictionaries
Dictionaries

• A dictionary is like an array, but it can be indexed using strings (or numbers, or tuples, or any immutable type)
  • the values used as indexes for a particular dictionary are called its *keys*
  • think of a dictionary as an unordered collection of *key : value* pairs
  • empty dictionary: {}

• It is an error to index into a dictionary using a non-existent key
Dictionaries

empty dictionary
Dictionaries

empty dictionary

populating the dictionary
• in this example, one item at a time
Dictionaries

empty dictionary

populating the dictionary
• in this example, one item at a time

looking up the dictionary (indexing)
Dictionaries

- empty dictionary
- populating the dictionary
  - in this example, one item at a time
- looking up the dictionary (indexing)
- looking at the dictionary
  - we can use this syntax to populate the dictionary too
empty dictionary
populating the dictionary
• in this example, one item at a time
looking up the dictionary (indexing)
looking at the dictionary
• we can use this syntax to populate the dictionary too
indexing with a key not in the dictionary is an error (**KeyError**)
Dictionaries

- initializing the dictionary
- in this example, several items at once
Dictionaries

- Initializing the dictionary
  - In this example, several items at once

- Getting a list of keys in the dictionary
  - Useful since it’s an error to index into a dictionary with a key that is not in it
We can use a **for** loop to iterate through a dictionary.
We can use a **for** loop to iterate through a dictionary

Notice that this iteration may not list the items in the dictionary in the same order as when they were inserted.
EXERCISE

```python
>>> crs_units = { 'csc 352': 3, 'csc 120': 4, 'csc 110': 4 }
>>> for crs in crs_units:
    print( "{0} : {1} units".format( crs, crs_units[crs] ) )

csc 110 : 4 units
csc 120 : 4 units
csc 352 : 3 units

How can we get the dictionary contents to be printed out in sorted order of the keys? (I.e., what goes in the box?)
```