15. Importing Modules

All Python programs can call a basic collection of functions called built-in functions, including the `print()`, `input()`, and `len()` functions you've seen before. Python also comes with a set of modules called the standard library. Each module is a file containing various related definitions, including object references functions and classes that can be embedded in your programs.

For example, the `math` module has mathematics-related functions, the `random` module has random number–related functions, and so on.
Before you can use the functions in a module, you must `import` the module with an import statement. In code, an `import` statement consists of the following:

- The `import` keyword
- The name of the module
- Optionally, more module names, as long as they are separated by commas

Once you import a module, you can use all the definitions of that module. Let's give it a try with the `random` module, which will give us access to the `random.randint()` function.
Enter this code into the file editor, and save it as *printRandom.py*:

```
import random
for i in range(5): print(random.randint(1, 10))
```

When you run this program, the output will look something like this:

```
4
1
8
4
1
```

The `random.randint()` function call evaluates to a random integer value between the two integers that you pass it. Since `randint()` is in the random module, you must first type `random`. in front of the function name to tell Python to look for this function inside the random module.

Here's an example of an import statement that imports four different modules:

```
import random, sys, os, math
```

Now we can use any of the functions in these four modules. We'll learn more about them later in these notes.

The `from import` Statements

An alternative form of the import statement is composed of the `from` key- word, followed by the module name, the `import` keyword, and a list of functions or classes defined in the module. Example:

```
from random import randint, choices
```

After this statement, calls to the functions `randint and choices` from `random` will not need the `random.` prefix.

You may use all names from a module `M` without the `M.` prefix by an import statement like this:

```
from M import *
```

This is not recommended because you could easily end up with name collisions ("namespace pollution"). The one situation where you will see this is with the *tkinter* GUI module:

```
from tkinter import *
```

If you find a module name too long you can use an `import as` statement:

```
import tkinter as tk
```

Now you can use the prefix `tk.` in place of `tkinter.`

16. Files and Paths

A *path separator* is a symbol used to join together directory names to form a path. For example, the Windows path separator is a backslash ('`\`') and the Unix and Mac separator is a forward slash '`/`'). The Windows separator is unfortunately also used for "escaping" unprintable characters. Thus '`\n`' is the line terminator and '`\t`' if the tab character. Thus if I had a directory named '`tasks`', it would be necessary to put two backslashes before the `t`; this indicates that the backslash should be treated as an ordinary character, not part of a special escaped character like the tab. So we end up with paths like `'usr\\bin\\spam'`.

Luckily, Python provides a way for you to write system independent code. The `os` module has an attribute `sep`. So the path `'usr\\bin\\spam` could be written as `os.sep.join('C:','Users', 'myDocs').`

A better approach would use the `os.path.join()` method:

Windows:

```
>>> import os
>>> os.path.join('usr',  'bin', 'spam')
```

```
'usr\\bin\\spam'
```

OS X or Linux:

```
>>> import os
>>> os.path.join('usr',  'bin', 'spam')
'usr/bin/spam'
```

The `os.path.join()` function is helpful if you need to create strings for filenames. These strings will be passed to several of the file-related functions introduced in these notes. For example, the following example joins names from a list of filenames to the end of a folder's name:

```
>>>  myFiles   =   ['accounts.txt',   'details.csv', 'invite.docx']
>>> for  filename  in myFiles:
       …    print(os.path.join('C:\\Users\\asweigart', filename))
    C:\Users\asweigart\accounts.txt
    C:\Users\asweigart\details.csv
    C:\Users\asweigart\invite.docx
```

Some useful os methods:

```
>>> import os
>>> os.getcwd()  # get current working directory
'C:\\Python34'
>>> os.chdir('C:\\Windows\\System32')
>>> os.getcwd()
'C:\\Windows\\System32'
```

Python will display an error if you try to change to a directory that does not exist.

```
>>> os.chdir('C:\\ThisFolderDoesNotExist')
Traceback (most recent call last):
File "<pyshell#18>", line 1, in <module>
os.chdir('C:\\ThisFolderDoesNotExist')
FileNotFoundError: [WinError 2] The system cannot find the file
specified: 'C:\\ThisFolderDoesNotExist'
```

Creating New Folders

You can create a new folder with the `os.mkdir` command.

```
>>> os.getcwd()
'/user/judy/docs'

>>> os.mkdir('pythonstuff')

>>> os.chdir('pythonstuff')  # no error – folder exists

>>> os.mkdir(os.path.join('adir', 'asubdir')) # adir/subdir
Traceback (most recent call last):
File "<pyshell#18>", line 1, in <module>
os.chdir('C:\\ThisFolderDoesNotExist')
FileNotFoundError: [Errno 2] No such file or directory: 'adir/asubdir'
```

The problem is that you can only create one directory with `mkdir`, not two (`adir` and `asubdir`). To create all folders on a path you should use os.makedirs.

Suppose there is no subdirectory of the current directory named delicious. Then the command

```
>>> os.makedirs('delicious/walnut/waffles')
```

will create the following folders, in order: `'delicious'`, `'delicious/walnut'` and `'delicious/walnut/waffles'`.

The `os.path` Module

The `os.path` module contains many helpful functions related to filenames and file paths. For instance, you've already used `os.path.join()` to build paths in a way that will work on any operating system. Since `os.path` is a module inside the os module, you can import it by simply running `import os`. Whenever your programs need to work with files, folders, or file paths, you can refer to the short examples in this section. The full documentation for the `os.path` module is on the Python website at http://docs.python.org/3/ library/os.path.html.

The `os.path` module provides functions for returning the absolute path of a relative path and for checking whether a given path is an absolute path.

Calling `os.path.abspath(path)` will return a string of the absolute path of the argument. This is an easy way to convert a relative path into an absolute one.

Calling `os.path.isabs(path)` will return True if the argument is an abso- lute path and False if it is a relative path.

Calling `os.path.relpath(path, start)` will return a string of a relative path from the start path to path. If start is not provided, the current working directory is used as the start path.

```
>>> os.path.abspath('.')
'/Users/ralph/courses2017fall/advpy2017fall/Tutorials'
```

```
>>> os.chdir('../..')
```

```
>>> os.getcwd()
'/Users/ralph/courses2017fall'
```

```
>>> os.path.abspath('./Tutorials')
'/Users/ralph/courses2017fall/Tutorials'
```

```
>>> os.path.isabs('.')
False
```

Calling `os.path.dirname(path)` will return a string of everything that comes before the last slash in the path argument. Calling `os.path.basename(path)` will return a string of everything that comes after the last slash in the path argument. The dir name and base name of a path are outlined below. If `pth = '/Users/ralph/courses2017fall/try.py'` then `os.path.dirname(pth)` is `'Users/ralph/courses2017fall'` and `os.path.dirname(pth)` is `'try.py'`.

If you need a path's dir name and base name together, you can just call `os.path.split()` to get a tuple value with these two strings, like so:

```
>>> os.path.split(pth)
('Users/ralph/courses2017', 'try.py')
```

Finding File Sizes and Folder Contents

Once you have ways of handling file paths, you can then start gathering information about specific files and folders.

Calling `os.listdir(path)` will return a list of filename strings for each file and folder in the path argument. (Note that this function is in the `os` module, not `os.path`.)

```
    >>> os.getcwd()
    '/Users/ralph/courses2017fall/advpy2017fall/tst'
    >>> os.listdir()
    ['rien', 'subd1']
    >>> os.chdir('..')
    >>> os.listdir('tst')
    ['rien', 'subd1']
    >>> os.listdir('tst/rien')
    Traceback (most recent call last):
      File "<stdin>", line 1, in <module>
    NotADirectoryError: [Errno 20] Not a directory: 'tst/rien'
    >>> os.listdir('tst/subd1')
    []
```

Calling `os.path.getsize(path)` will return the size in bytes of the file in the path argument.

```
    >>> os.path.getsize('tst/rien')
    5
```

If I want to find the total size of all the files in a directory, I can use `os.path.getsize()` and `os.listdir()` together.

```
>>> totalSize   =  0
>>> for  filename  in os.listdir('C:\\Windows\\System32'):
        (totalSize  += os.path.getsize(os.path.join(
                        'C:\\Windows\\System32',  filename)))
>>> print(totalSize)
1117846456
```

Checking Path Validity

Many Python functions will crash with an error if you supply them with a path that does not exist. The `os.path` module provides functions to check whether a given path exists and whether it is a file or folder.

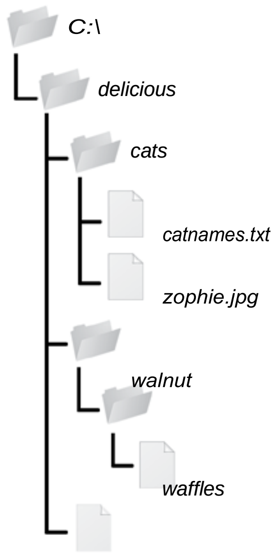
Calling `os.path.exists(path)` will return `True` if the file or folder referred to in the argument exists and will return `False` if it does not exist.

Calling `os.path.isfile(path)` will return `True` if the path argument exists and is a file and will return `False` otherwise.

Calling `os.path.isdir(path)` will return `True` if the path argument exists and is a folder and will return `False` otherwise.

Walking a Directory Tree

Say you want to rename every file in some folder and also every file in every subfolder of that folder. That is, you want to walk through the directory tree, touching each file as you go. Writing a program to do this could get tricky; fortunately, Python provides a function to handle this process for you. Let's look at the C:\delicious folder with its contents, shown below.



Here is an example program that uses the `os.walk()` function on the directory tree above.

```
import os

for folderName, subfolders, filenames in os.walk('C:\\delicious'):
    print('The current folder is ' + folderName)

for subfolder in subfolders:
    print('SUBFOLDER OF ' + folderName + ': ' + subfolder)

for filename in filenames:
    print('FILE INSIDE ' + folderName + ': '+ filename) print('')
```

The os.walk() function is passed a single string value: the path of a folder. You can use os.walk() in a for loop statement to walk a directory tree, much like how you can use the range() function to walk over a range of numbers. Unlike range(), the os.walk() function will return three values on each iteration through the loop:

1. A string of the current folder's name
2. A list of strings of the folders in the current folder
3. A list of strings of the files in the current folder

(By current folder, I mean the folder for the current iteration of the for loop. The current working directory of the program is not changed by `os.walk()`.)

When you run this program, it will output the following:

```
The current folder is C:\delicious
SUBFOLDER OF C:\delicious: cats
SUBFOLDER OF C:\delicious: walnut
FILE INSIDE C:\delicious: spam.txt

The current folder is C:\delicious\cats
FILE INSIDE C:\delicious\cats: catnames.txt
FILE INSIDE C:\delicious\cats: zophie.jpg

The current folder is C:\delicious\walnut
SUBFOLDER OF C:\delicious\walnut: waffles

The current folder is C:\delicious\walnut\waffles
FILE INSIDE C:\delicious\walnut\waffles: butter.txt.
```

Since `os.walk()` returns lists of strings for the subfolder and filename variables, you can use these lists in their own for loops. Replace the `print()` function calls with your own custom code. (Or if you don't need one or both of them, remove the `for` loops.)

You are now ready for the BigFiles assignment.

17. Sets

A set is like a dictionary with its values thrown away, leaving only the keys. It is a mutable unordered collection of distinct objects. Like dictionaries, sets are implemented using hashing and thus their elements must be *hashable*. The tuple (1, 2, 3, 4} can be a set element, but the tuple (1,[2,3],4} cannot be an element of a set. You use a set when you only want to know that something exists, and nothing else about it. You use a dictionary if you want to attach some information to the key as a value.

Create with `set()`

To create a set, you use the set() function or enclose one or more comma-separated values in curly brackets, as shown here:

```
>>> empty_set = set()
>>> empty_set
set()
>>> even_numbers = {0, 2, 4, 6, 8}
>>> even_numbers
{0, 8, 2, 4, 6}
>>> odd_numbers = {1, 3, 5, 7, 9}
>>> odd_numbers
{9, 3, 1, 5, 7}
```

As with dictionary keys, sets are unordered..

Even though both are enclosed by curly braces ({ and }), a set is just a sequence of values, and a dictionary is one or more *key : value* pairs

Because `[]` creates an empty list, you might expect `{}` to create an empty set. Instead, `{}` creates an empty dictionary. That's also why the interpreter prints an empty set as `set()` instead of `{}`. Why? ***Dictionaries were in Python first and took possession of the curly brackets***.

Convert from Other Data Types with `set()`

You can create a set from a list, string, tuple, or dictionary, discarding any duplicate values. First, let's take a look at a string with more than one occurrence of some letters:

```
>>> set( 'letters' )
{'l', 'e', 't', 'r', 's'}
```

Notice that the set contains only one 'e' or 't', even though 'letters' contained
two of each.

Now, let's make a set from a list:

```
>>> set( ['Dasher', 'Dancer', 'Prancer', 'Mason-Dixon'] )
{'Dancer', 'Dasher', 'Prancer', 'Mason-Dixon'}
```

This time, a set from a tuple:

```
>>> set( ('Ummagumma', 'Echoes', 'Atom Heart Mother') )
{'Ummagumma', 'Atom Heart Mother', 'Echoes'}
```

When you give set() a dictionary, it uses only the keys:

```
>>> set( {'apple': 'red', 'orange': 'orange', 'cherry': 'red'} )
{'apple', 'cherry', 'orange'}
```

Sets always contain unique items—adding duplicate items is safe but pointless. For example, these three sets are the same:

```
set("apple"), set("aple"), and {'e','p', 'a', 'l'}.
```

In view of this, sets are often used to eliminate duplicates. For example, if `x` is a list of strings, then after executing `x = list(set(x))`all of `x`'s strings will be unique—and in an arbitrary order.

You may use the **`in`** operator to test for membership:

```
>>> s = {'ab','cd',2,(3,4)}
>>> 2 in s
True
>>> 'ab' not in s
False
```

One common use case for sets is when we want fast membership testing.

Example: give the user a usage message if they don’t enter any command-line arguments; or they enter an argument of “-h” or “--help”:

```
if (len(sys.argv) == 1 or
     sys.argv[1] in {"-h", "--help"}):
```

Another common use case for sets is to ensure that we don't process duplicate data.

Example:

we have an iterable (such as a list), containing the IP addresses from a web server's log files
we wanted to perform some processing, once for each unique address

Assuming that the IP addresses are hashable and the IP addresses are in the iterable `ips`, we could use either of the code segments below.

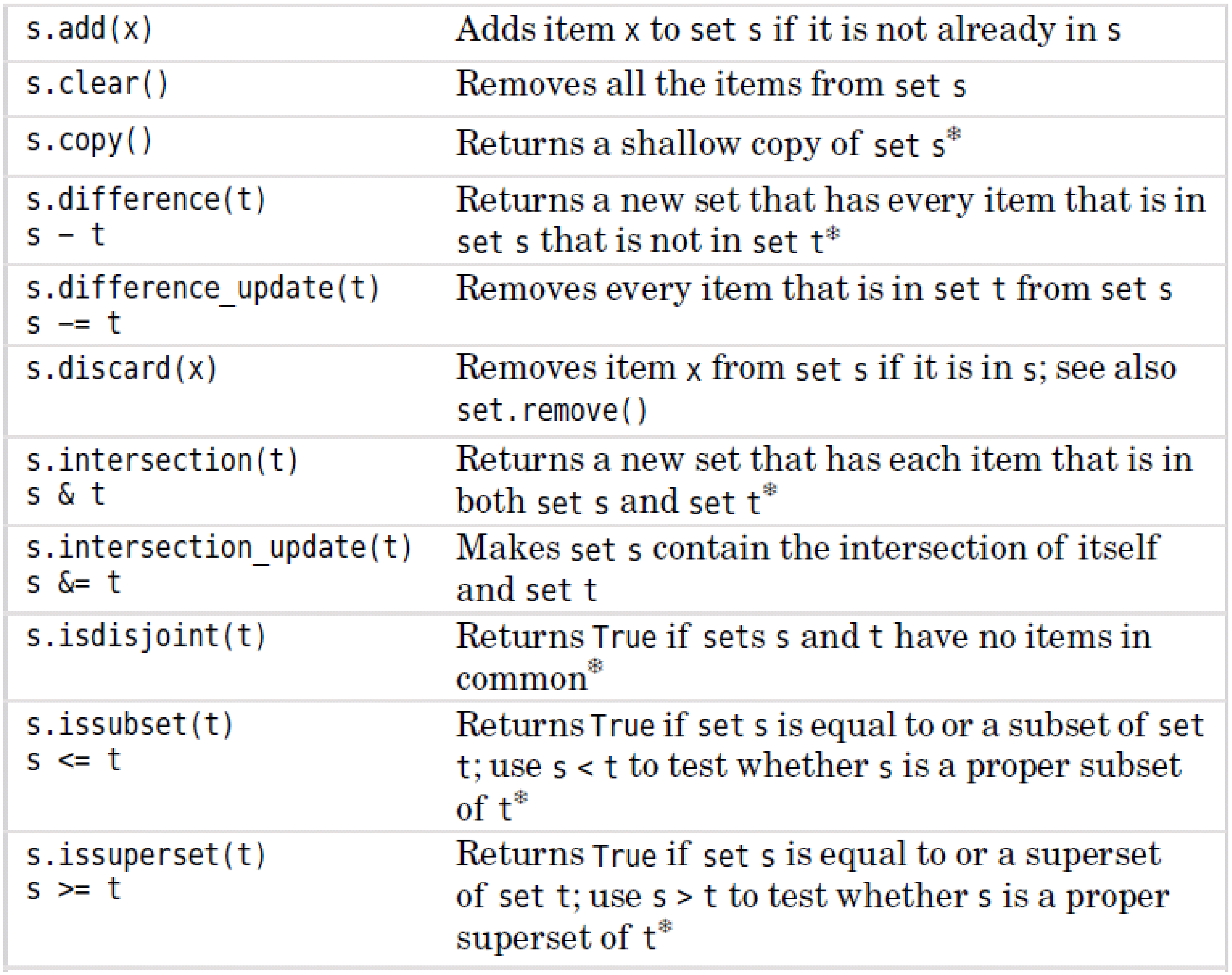
```
# version 1
seen = set()
for ip in ips:
    if ip not in seen:
        seen.add(ip)
        process_ip(ip)


# version 2
for ip in set(ips):
    process_ip(ip)
```

The first snippet processes the IP addresses in the order they are encountered in the `ips` iterable. The second snippet processes them in an arbitrary order.

Set Operations

The comparisions == and != have their normal meanings. For other set operators see the two tables below.



`s.add(x)`	Adds item `x` to set `s` if it is not already in `s`
`s.clear()`	Removes all the items from set `s`
`s.copy()`	Returns a shallow copy of set `s`*

`s.difference(t)`
`s - t`
	Returns a new set that has every item that is in set `s` that is not in set `t`*

`s.difference_update(t)`
`s -= t`
	Removes every item that is in set `t` from set `s`

`s.discard(x)`	Removes item `x` from set `s` if it is in `s`; see also `set.remove()`

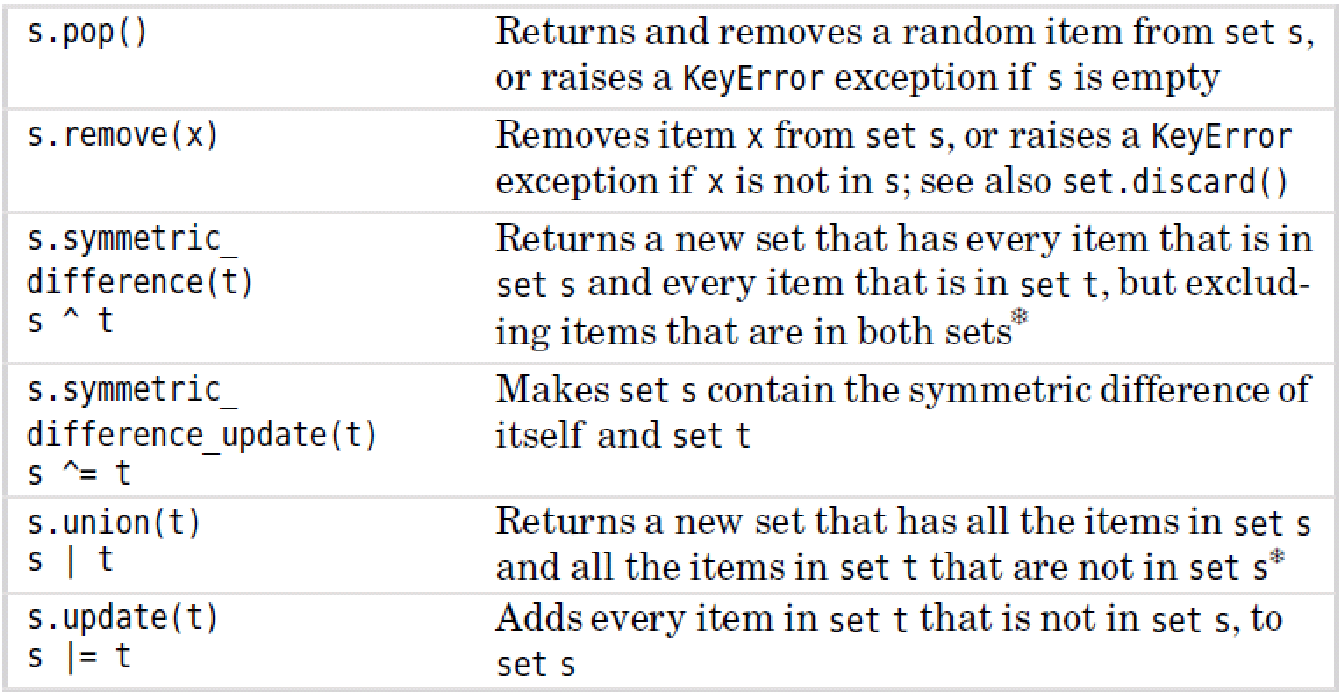
`s.intersection(t)`
`s & t`
	Returns a new set that has each item that is in both set `s` and set `t`*

`s.intersection_update(t)`
`s &= t`
	Makes set `s` contain the intersection of itself and set `t`

`s.isdisjoint(t)`	Returns `True` if sets `s` and `t` have no items in common*

`s.issubset(t)`
`s <= t`
	Returns `True` if set `s` is equal to or a subset of set `t`; use `s < t` to test whether `s` is a proper subset of `t`*

`s.issuperset(t)`
`s >= t`
	Returns `True` if set `s` is equal to or a superset of set `t`; use `s > t` to test whether `s` is a proper superset of `t`*



`s.pop()`	Returns and removes a random item from set `s`, or raises a `KeyError` exception if `s` is empty
`s.remove(x)`	Removes item `x` from set `s`, or raises a `KeyError` exception if `x` is not in `s`; see also `set.discard()`
`s.symmetric_difference(t)` `s ^ t`	Returns a new set that has every item that is in set `s` and every item that is in set `t`, but excluding items that are in both sets*
`s.symmetric_difference_update(t)` `s ^= t`	Makes set `s` contain the symmetric difference of itself and set `t`
`s.union(t)` `s | t`	Returns a new set that has all the items in set `s` and all the items in set `t` that are not in set `s`*
`s.update(t)` `s |= t`	Adds every item in set `t` that is not in set `s`, to set `s`

All the "update" methods – **set.update(), set.intersection_update()**, etc. – accept any iterable as their argument. The equivalent operator versions (|=, &=, etc.) require both of their operands to be sets.

The ***frozenset*** type is the immutable version of the set type. Some, but clearly not all of the set operations apply to frozensets.

18. Comprehensions

Comprehensions are one the most used features in Python. They are syntactic constructs used to create lists, sets or dictionaries.

List comprehensions

Small lists are often created using list literals, but longer lists are usually created programmatically. For a list of integers, we can use `list(range(n))`, or if we just need an integer iterator, `range()` is sufficient, but for other lists using a `for`…`in` loop is very common. Suppose, for example, that we wanted to produce a list of the leap years in a given range. We might start out like this:

```
leaps = []
for year in range(1900, 1940):
   if (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0):
      leaps.append(year)
```

A *list comprehension* is an expression and a loop with an optional condition enclosed in brackets where the loop is used to generate items for the list, and where the condition can filter out unwanted items. The simplest form of a list comprehension is this:

[item for item in iterable]

This will return a list of every item in the iterable, and is semantically no different from list(iterable). Two things that make list comprehensions more interesting and powerful are that we can use expressions, and we can attach a condition—this takes us to the two general syntaxes for list comprehensions:

[expression for item in iterable]

[expression for item in iterable if condition]

The second syntax is equivalent to:

```
temp = []
for item in iterable:
   if condition:
      temp.append(expression)
```

Normally, the expression will either be or involve the item. Of course, the list comprehension does not need the temp variable needed by the `for … in` loop version.

Now we can rewrite the code to generate the leaps list using a list comprehension. We will develop the code in three stages. First we will generate a list that has all the years in the given range:

```
leaps = [y for y in range(1900, 1940)]
```

This could also be done using leaps = list(range(1900, 1940)). Now we'll add a simple condition to get every fourth year:

```
leaps = [y for y in range(1900, 1940) if y % 4 == 0]
```

Finally, we have the complete version:

```
leaps = [y for y in range(1900, 1940)

if (y % 4 == 0 and y % 100 != 0) or (y % 400 == 0)]
```

Using a list comprehension in this case reduced the code from four lines to two—a small savings, but one that can add up quite a lot in large projects.

Since list comprehensions produce lists, that is, iterables, and since the syntax for list comprehensions requires an iterable, it is possible to nest list comprehensions. This is the equivalent of having nested `for…in` loops. For example, if we wanted to generate all the possible clothing label codes for given sets of sexes, sizes, and colors, but excluding labels for the full-figured females whom the fashion industry routinely ignores, we could do so using nested `for …in` loops:

```
codes = []
for sex in "MF": # Male, Female
   for size in "SMLX": # Small, Medium, Large, eXtra large
      if sex == "F" and size == "X":
         continue
      for color in "BGW": # Black, Gray, White
         codes.append(sex + size + color)
```

This produces the 21 item list, `['MSB', 'MSG',…, 'FLW']`.

The list `['MSB', 'MSG',…, 'FLW']` created in the above code using `for…in` loops can be constructed in just a couple of lines using a list comprehension:

```
codes = [s + z + c for s in "MF" for z in "SMLX" for c in "BGW" if
         not (s == "F" and z == "X")]
```

Here, each item in the list is produced by the expression `s + z + c`. Also, we have used subtly different logic for the list comprehension where we skip invalid sex/size combinations in the innermost loop, whereas the nested `for…in` loops version skips invalid combinations in its middle loop. Any list comprehension can be rewritten using one or more `for…in` loops.

If the generated list is very large, it may be more efficient to generate each item as it is needed rather than produce the whole list at once. This can be achieved by using a ***generator*** rather than a list comprehension. We discuss generators later.

Set Comprehensions

In addition to creating sets by calling set(), or by using a set literal, we can also create sets using set comprehensions. A set comprehension is an expression and a loop with an optional condition enclosed in braces. Like list comprehensions, two syntaxes are supported:

{expression for item in iterable}
{expression for item in iterable if condition}

We can use these to achieve a filtering effect (providing the order doesn't matter). Here is an example:

```
html = {x for x in files if x.lower().endswith((".htm", ".html"))}
```

Given a list of filenames in files, this set comprehension makes the set html hold only those filenames that end in `.htm` or `.html`, regardless of case.

Just like list comprehensions, the iterable used in a set comprehension can itself be a set comprehension (or any other kind of comprehension), so quite sophisticated set comprehensions can be created.

Dictionary Comprehensions

A dictionary comprehension is an expression and a loop with an optional condition enclosed in braces, very similar to a set comprehension. Like list and set comprehensions, two syntaxes are supported:

> *{keyexpression: valueexpression for key, value in iterable}*
> *{keyexpression: valueexpression for key, value in iterable if condition}*

Here is how we could use a dictionary comprehension to create a dictionary where each key is the name of a file in the current directory and each value is the size of the file in bytes:

```
file_sizes = {name: os.path.getsize(name) for name in
              os.listdir(".")}
```

We can avoid directories and other non-file entries by adding a condition:

```
file_sizes = {name: os.path.getsize(name) for name in os.listdir(".")
              if os.path.isfile(name)}
```

A dictionary comprehension can also be used to create an inverted dictionary. For example, given dictionary d, we can produce a new dictionary whose keys are d’s values and whose values are d’s keys:

```
inverted_d = {v: k for k, v in d.items()}
```

The resultant dictionary can be inverted back to the original dictionary if all the original dictionary’s values are unique—but the inversion will fail with a TypeError being raised if any value is not hashable.

Just like list and set comprehensions, the iterable in a dictionary comprehension can be another comprehension, so all kinds of nested comprehensions are possible.

19. Exceptions

You should have already encountered the concept of exceptions in your Data Structures class. In this section, we explain python exceptions.

Exception handling

Python exceptions are used in try-except blocks (as opposed to try-catch used in C++).

```
try:
   try_suite
except exception_group1 as variable1:
   except_suite1
       …
except exception_groupN as variableN:
   except_suiteN
else:
   else_suite
finally:
   finally_suite
```

There must be at least one except block, but both the else and the finally blocks are optional.

The else block’s suite is executed when the try block’s suite has finished normally—but it is not executed if an exception occurs.

If there is a finally block, it is always executed at the end. Each except clause’s exception group can be a single exception or a parenthesized tuple of exceptions.

For each group, the as variable part is optional. If used, the variable contains the exception that occurred, and can be accessed in the exception block’s suite.

If an exception occurs in the try block’s suite, each except clause is tried in turn. If the exception matches an exception group, the corresponding suite is executed.

To match an exception group, the exception must be of the same type as one of the exception types listed in the group, or a subclass of a group’s exception type.

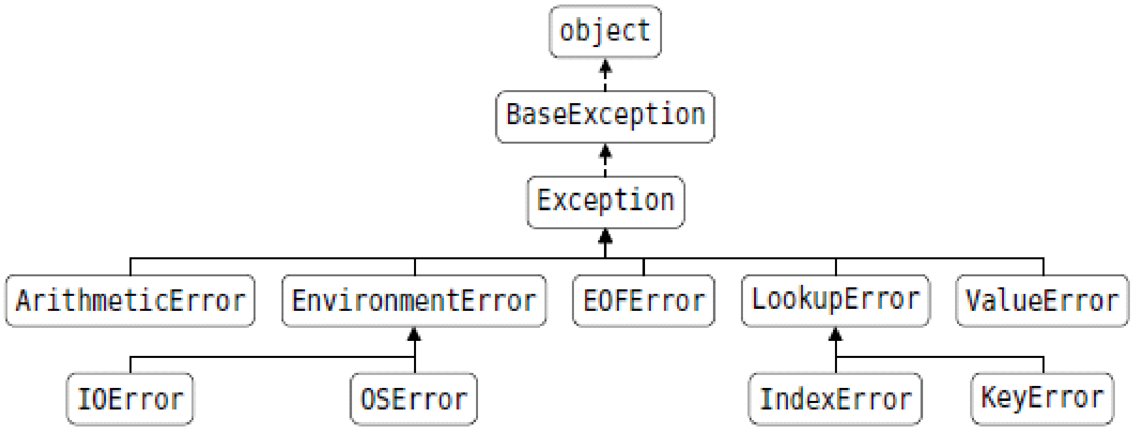
Exception Example

If a `KeyError` exception occurs in a dictionary lookup, the first except clause that has an `Exception` class will match since `KeyError` is an (indirect) subclass of `Exception`.

If no group lists Exception (as is normally the case), but one did have a `LookupError`, the `KeyError` will match, because `KeyError` is a subclass of `LookupError`.

And if no group lists `Exception` or `LookupError`, but one does list `KeyError`, then that group will match.

The following diagram shows the exception hierarchy.



Example: Incorrect Use

```
try:
  x = d[5]
except LookupError:                        # WRONG ORDER
  print("Lookup error occurred")
except KeyError:
   print("Invalid key used")
```

If dictionary d has no item with key 5, we want the most specific exception, KeyError, to be raised, rather than the more general LookupError exception.

But here, the `KeyError` except block will never be reached. If a `KeyError` is raised, the `LookupError` except block will match it because `LookupError` is a base class of `KeyError`: `LookupError` appears higher than `KeyError` in the exception hierarchy

When we use multiple except blocks, we must always order them from most specific (lowest in the hierarchy) to least specific (highest in the hierarchy)

Bad Practice

```
try:
   x = d[k/n]
except Exception:
   print("Something happened")
```

The above is usually bad practice since this will catch all exceptions and could easily mask logical errors in our code. In this example, we might have intended to catch `KeyErrors` But, if `n` is `0`, we will unintentionally—and silently—catch a `ZeroDivisionError` exception.

It is also possible to write `except:`

That is, to have no exception group at all. An `except` block like this will catch any exception, including those that inherit `BaseException` but not `Exception`

This has the same problems as using except `Exception`, only worse, and should never normally be done.

Stack Trace

If none of the `except` blocks matches an exception, Python will work its way up the call stack looking for a suitable exception handler. If none is found the program will terminate and print the exception and a traceback on the console.

If no exceptions occur, any optional `else` block is executed. And in all cases—that is, if no exceptions occur, if an exception occurs and is handled, or if an exception occurs that is passed up the call stack—any `finally` block's suite is always executed

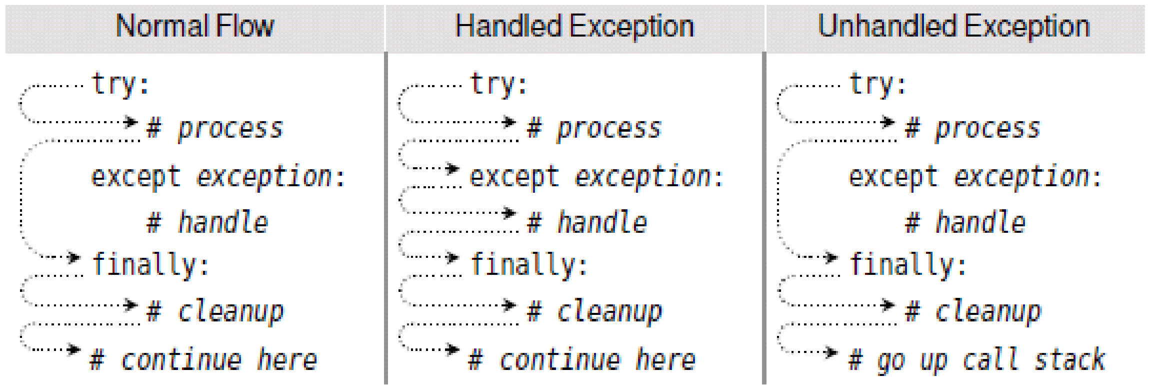
`finally` Block

If no `exception` occurs, or if an `exception` occurs and is handled by one of the `except` blocks, the `finally` block's suite is executed at the end.

If an `exception` occurs that doesn't match, first the `finally` block's suite is executed, and then the exception is passed up the call stack.

This guarantee of execution can be very useful when we want to ensure that resources are properly released.

The general `try` …`except` …`finally` block control flows:



Example

We can use exceptions to write our a function that does the same thing as the `list.find()` method.

```
def list_find(lst, target):
   try:
      index = lst.index(target)
   except ValueError:
      index = -1
   return index
```

Simple try … finally Block

```
try:
   try_suite
finally:
   finally_suite
```

No matter what happens in the try block's suite (apart from the computer or program crashing!), the finally block's suite will be executed.

```
def read_data(filename):
   lines = []
   fh = None
   try:
      fh = open(filename, encoding="utf8")
      for line in fh:
         if line.strip():
            lines.append(line)
   except (IOError, OSError) as err:
      print(err)
      return []
   finally:
      if fh is not None:
         fh.close()
   return lines
```

No matter what happens, the file will be safely closed.

In the `read_data()` function, we could have written the except clause slightly less verbosely:

```
 except EnvironmentError as err:
   print(err)
   return []
```

This works because `EnvironmentError` is the base class for both `IOError` and `OSError`

Note: The `with` statement can be used to achieve a similar effect to using a `try` …`finally block`.

Raising Exceptions

Exceptions provide a useful means of changing the flow of control. We can take advantage of this either by using the built-in exceptions, or by creating our own, raising either kind when appropriate.

There are three syntaxes for raising exceptions:

```
     raise exception(args)
     raise exception(args) from original_exception
     raise
```

When the first syntax is used the exception should be either one of the built-in exceptions, or a custom exception that is derived from `Exception`. If we give the exception some text as its argument, this text will be output if the exception is printed when it is caught.

Custom Exceptions

Custom exceptions are custom data types (classes):

```
     class exceptionName(baseException): pass
```

The base class should be `Exception` or a class that inherits from `Exception`.

One use of custom exceptions is to break out of deeply nested loops. For example, suppose we have a table object and the table object holds records (rows), records hold fields (columns) and fields have multiple values (items). Then we could search for a particular value with code like that shown below.

Here is the code that does not use exceptions:

```
found = False
for row, record in enumerate(table): # see the pocket reference, p137
   for column, field in enumerate(record):
      for index, item in enumerate(field):
         if item == target:
            found = True
            break
      if found:
         break
   if found:
      break
if found:
   print("found at ({0}, {1}, {2})".format(row, column,
                                           index))
else:
   print("not found")
```

Below is the code using a custom exception.

```
class FoundException(Exception): pass

 try:
   for row, record in enumerate(table):
      for column, field in enumerate(record)
         for index, item in enumerate(field):
            if item == target:
               raise FoundException()

 except FoundException:
   print("found at ({0}, {1}, {2})".format(row, column,
                                           index))
else:
   print("not found")
```

You should download and study the checktags.py file. Pay attention to how various conditions are checked and custom exceptions raised as needed. And how these exceptions are caught later in the code. This is an important example, so do not skip this!

20. Sorting and Reversing

Reversing: The built-in polymorphic function `reversed()` takes any iterable as an argument and produces an object whose type is a special kind of iterator (a "reversed iterator", or just a "reversed"). As you would expect, it iterates over the iterable from last to first. And, of course, you can construct a corresponding list or tuple by using the type name as a conversion function.

Here are some examples:

```
>>> L = [2,3,4]
>>> t = (5,6,7)
>>> type(reversed(L))
<class 'list_reverseiterator'>
>>> type(reversed(t))
<class 'reversed'>
>>> s = 'hat'
>>> type(reversed(s))
<class 'reversed'>
>>> list(reversed(L))
[4, 3, 2]
>>> tuple(reversed(t))
(7, 6, 5)
>>> str(reversed(s))
'<reversed object at 0x7f130a96e160>'
>>> ''.join(reversed(s))
'tah'
```

The conversions are all as expected except for the `str` object. The reason is that `str(x)` returns the string representation of `x` for any built-in type. So instead we used the join operator with the empty string as the "glue".

The `list` class has a method for reversing a list in place:

```
>>> K = [2,3,4,5]
>>> K
[2, 3, 4, 5]
>>> K.reverse()
>>> K
[5, 4, 3, 2]
```

There is no `reverse` *method* for tuples or strings since they are immutable.

Sorting

The built-in polymorphic function `sorted()` takes any iterable as an argument. It creates and returns a list of the elements of the iterable in sorted order. If there are elements in the iterable that are not comparable, a `TypeError` exception is raised.

```
>>> L = [3,1,2]
>>> sorted(L)
[1, 2, 3]
>>> L
[3, 1, 2]
>>> t = (3,1,2)
>>> sorted(t)
[1, 2, 3]
>>> s = 'bac'
>>> sorted(s)
['a', 'b', 'c']
>>> K = ['hi',2,1.5]
>>> sorted(K)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'int' and 'str'
```

The `sorted()` function takes an additional optional argument `key` of type function that can be used to change the sorting order. Instead of directly comparing elements `x` and `y`, it compares `key(x)` and `key(y)`.

```
>>> L = ['Hi','hello','Oh no','Apple', 'apt']
>>> sorted(L)
['Apple', 'Hi', 'Oh no', 'apt', 'hello']
>>> sorted(L,key=str.lower)
['Apple', 'apt', 'hello', 'Hi', 'Oh no']
>>> t = (3,-1,2,0,-4)
>>> sorted(t)
[-4, -1, 0, 2, 3]
>>> sorted(t,key=abs)
[0, -1, 2, 3, -4]
```

Note the `key` argument in the last example is the function object `abs`. If you had written abs(), it would be a functions *call*, not a function *object*.

The `list` class has a `sort` method that modifies the list in place.

```
>>> L = [4,3,5,1]
>>> L.sort()
>>> L
[1, 3, 4, 5]
>>> K = ['Hi','hello','Oh no','Apple', 'apt']
>>> K.sort(key=str.lower)
>>> K
['Apple', 'apt', 'hello', 'Hi', 'Oh no']
```

21. Sequence Unpacking

Although we can use the slice operator to access items in a list, in some situations we want to take two or more pieces of a list in one go. This can be done by *sequence unpacking.*

Any iterable (lists, tuples, etc.) can be unpacked using the sequence unpacking operator, an asterisk or star (*). When used with two or more variables on the left-hand side of an assignment, one of which is preceded by `*`, items are assigned to the variables, with all those left over assigned to the starred variable.

Examples

```
>>> first, *rest = [9, 2, -4, 8, 7]
>>> first, rest
(9, [2, -4, 8, 7])
>>> first,*mid,last = "Charles Philip Arthur George Windsor".split()
>>> first, mid, last
('Charles', ['Philip', 'Arthur', 'George'], 'Windsor')
>>> *directories, executable = "/usr/local/bin/gvim".split("/")
>>> directories, executable
(['', 'usr', 'local', 'bin'], 'gvim')
```

When the sequence unpacking operator is used like this, the expression *rest, and similar expressions, are called *starred expressions*

Python also has a related concept called *starred arguments*. For example, if we have the following function that requires three arguments:

```
def product(a, b, c):
  return a * b * c # here, * is the multiplication operator
```

we can call it with three arguments, or by using starred arguments:

```
>>> product(2, 3, 5)
30
>>> L = [2, 3, 5]
>>> product(*L)
30
>>> product(2, *L[1:])
30
```

There is never any syntactic ambiguity regarding whether operator * is the multiplication or the sequence unpacking operator.

When it appears on the left-hand side of an assignment it is the unpacking operator. When it appears elsewhere (e.g., in a function call) it is

- the unpacking operator when used as a unary operator
- the multiplication operator when used as a binary operator.

Tuples may also be unpacked. Note that when you use two variables separated by a comma, it will be treated as a tuple. Note the simple way to swap variable values (actually, references!).

```
>>> 2,3
(2, 3)
>>> a, b = (2,3)
>>> a
2
>>> b
3
>>> a,b = b,a
>>> a
3
>>> b
2
```