Today

- File I/O from last time
  - Slides 38-48
- The `os` and `sys` modules
- The `exec(…)` BIF
- Confirming parameter types
  - Revisit raising exceptions
- Passing by reference
- Lists of lists and dictionaries
- Finding minimums and maximums
- Timing code execution

Slides courtesy of Dr. Alan McLeod

`os` Module

- Behind the scenes, this module loads a module for your particular operating system
- Regardless of your actual OS, Python imports `os`
  - See Section 15.1 in the Python Library Reference
- Lots of goodies, particularly file system utilities
  - *e.g.*, `os.sep` is the directory separator for your OS
- The next few slides have a selection of file-related functions

Slides courtesy of Dr. Alan McLeod

`os` Module - Cont.

- `remove(…)` Deletes a file
- `rename(…)` Renames a file
- `walk(…)` Generates filenames in a directory tree (generator object)
- `chdir(…)` Changes the working directory
- `chroot(…)` Changes root directory of current process in Unix
- `listdir(…)` Lists files and folders in a directory

Slides courtesy of Dr. Alan McLeod

`os` Module - Cont.

- `getcwd()` Gets the current working directory
- `mkdir(…)` Creates a directory
- `rmdir(…)` Removes a directory
- `access(…)` Verify permission modes
- `chmod(…)` Changes permission modes

Slides courtesy of Dr. Alan McLeod

`os.path` Module

- `basename(…)` Returns the name of a file
- `dirname(…)` Returns the name of a directory
- `join(…)` Joins a directory and a filename
- `split(…)` Splits a path into a directory and a filename (a tuple)
- `splitext(…)` Returns filename and extension as tuple
- `getatime(…)` Returns last file access time
- `getctime(…)` Returns file creation time

Slides courtesy of Dr. Alan McLeod

`os.path` Module – Cont.

- `getmtime(…)` Returns file modification time
- `getsize(…)` Returns file size in bytes
- `exists(…)` Does file or directory name exist?
- `isdir(…)` Is this a directory name and does it exist?
- `isfile(…)` Is this a file and does it exist?

Slides courtesy of Dr. Alan McLeod

`os` Module - Demo

- LargeFileSearch.py
- Uses a recursive directory search
  - Don't worry about this technique
  - You are not responsible for knowing how to use recursion … yet

Slides courtesy of Dr. Alan McLeod

`os` Module - Cont.

- The `os` module also has many commands that allow you to run other non-Python code and programs from within your program
  - *e.g.*, `os.system()` allows you to run a system command (such as a DOS command)

Slides courtesy of Dr. Alan McLeod

Aside – the `exec()` BIF

- Also in `os` module
- Can execute Python code if it is supplied to the BIF as a string
  - The string could come from a file, for example
- Demo: RemoteFileExecution.py

Slides courtesy of Dr. Alan McLeod

`sys` Module

- See Section 27.1 in the Python Library Reference
- Contains more system functions and attributes
  - A small sampling is provided here
- `argv`
  - A list of all command line parameters sent to the interpreter
- `builtin_module_names`
  - A list of all built-in modules
- `exit(0)`
  - Immediately exits a Python program giving a zero exit status code

Slides courtesy of Dr. Alan McLeod

`sys` Module - Cont.

- `getwindowsversion()`
  - Returns a tuple consisting of *major*, *minor*, *build*, *platform*, and *service pack status*
- `path`
  - A list of the module search paths used by Python
- `platform`
  - The current OS platform
- `prefix`
  - The folder where Python is located
- `version`
  - The version of Python being used

Slides courtesy of Dr. Alan McLeod

The `isinstance(…)` BIF

- When you get an argument value mapped to a parameter, how do you know it is the right type?
- The function assumes that the function is invoked with the proper types
- But should you check, and how can you?
- Suppose you have a parameter called **`param`** and it is supposed to be a string

```
isinstance(param, str)
```

will return **`True`**, **`False`** otherwise

Slides courtesy of Dr. Alan McLeod

`isinstance(…)`- Cont.

- What other types can you check?
  - **`bool`**
  - **`int float complex`**
  - **`str`**
  - **`list tuple set dict range`**
- Many other types exist in Python
  - You can have an object type as well

Slides courtesy of Dr. Alan McLeod

Raising Exceptions - Revisited

- So, what do you in your function if your parameter type is not correct?
- Demo: RaiseExceptionIsInstance.py

Slides courtesy of Dr. Alan McLeod

Passing by Reference

- Can a function change something in its parameter list and have the change stay (or “stick”) when the function is done?
- What kinds of parameters can be changed and how?
- Demo: TestPassingByReference.py

Slides courtesy of Dr. Alan McLeod

Passing by Reference - Observations

- Immutable objects do not stay changed outside the function
  - That’s the `int`, the `string` and the `tuple`
  - All you can do inside the function is assign a new value to the parameter
- Re-assigning a mutable object does not change it
- However, some actions allow the changes to stay after the function is complete
  - Element-by-element changes using the slice operator
  - Invoking a method belonging to a list

Slides courtesy of Dr. Alan McLeod

Passing by Reference - Cont.

- When you pass a list (or any object) into a function, you do not re-create the entire structure inside the function
  - That would be wasteful and time-consuming!
- Instead you just pass a *reference* (a memory address, or "pointer") into the function
- If the object is mutable, and its elements are changed or deleted inside the function, then that change is made to the structure created outside the function

Slides courtesy of Dr. Alan McLeod

Passing by Reference - Cont.

- We can take advantage of being able to pass mutable objects (especially lists) by reference to simplify code!
- This also gives you a way to get more than one thing out of a function without having to return a tuple of lists
  - **But**, if you are doing this maybe your function is not just doing one thing!
  - **And**, returning multiple things through the parameter list can make for confusing code

Slides courtesy of Dr. Alan McLeod

Lists of Lists

- We know a list can hold anything
  - The elements do not even have to be the same type

```
ex1 = [1, 4.0, 'abc', 2, 'hello!']
```

- So, there is no reason that an element cannot be another list (or a tuple, or some other collection)

```
ex2 = [4.5, [1, 2, 'abc'], 7, 'hello']
```


Slides courtesy of Dr. Alan McLeod

Lists of Lists - Example

```
>>> for value in ex2:
  print(value)


4.5
[1, 2, 'abc']
7
hello
```


Slides courtesy of Dr. Alan McLeod

Lists of Lists - Cont.

- How can I display the elements in the list at position 1?

```
>>> for value in ex2[1]:
  print(value)

1
2
abc
```


Slides courtesy of Dr. Alan McLeod

Lists of Lists - Cont.

- Nothing new!
- How do I access just the **`'abc'`** string inside the list at position 1?

```
>>> ex2[1][2] = 'wxyz'
>>> ex2
[4.5, [1, 2, 'wxyz'], 7, 'hello']
```


Slides courtesy of Dr. Alan McLeod

Lists of Lists - Cont.

- A list of lists can be used represent tabular data
- Suppose you wish to store people's names, ages and student numbers?

```
ex3 = [['Sam', 18, 4445555], ['Boris', 21,
  5554444], ['Ben', 19, 5445444]]
```

- You could do it this way, or (better yet) use a dictionary

Slides courtesy of Dr. Alan McLeod

Dictionaries – An Example

```
>>> name1 = {'name':'Sam', 'age':18, 'SN':4445555}
>>> name2 = {'name':'Boris', 'age':21, 'SN':5554444}
>>> name3 = {'name':'Ben', 'age':19, 'SN':5445444}
>>> allNames = [name1, name2, name3]
>>> allNames
[{'age': 18, 'name': 'Sam', 'SN': 4445555}, {'age':
  21, 'name': 'Boris', 'SN': 5554444}, {'age': 19,
  'name': 'Ben', 'SN': 5445444}]

>>> allNames[2]['age']
19
>>> allNames[1]['name']
'Boris'
```


Slides courtesy of Dr. Alan McLeod

Dictionaries – Better Yet

```
>>> allNames = {}
>>> allNames['Sam'] = {'age':18, 'SN':4445555}
>>> allNames['Boris'] = {'age':21, 'SN':5554444}
>>> allNames['Ben'] = {'age':19, 'SN':5445444}
>>> allNames
{'Boris': {'age': 21, 'SN': 5554444}, 'Ben': {'age':
  19, 'SN': 5445444}, 'Sam': {'age': 18, 'SN':
  4445555}}

>>> allNames['Boris']['age']
21
```


Slides courtesy of Dr. Alan McLeod

Dictionaries

- Indexing in a dictionary or **`dict`** is done using key names, not sequential numeric index values
- It is a “mapping” type data structure
- It does not matter what the order of the `key:value` pairs is

Slides courtesy of Dr. Alan McLeod

Dictionaries – Adding Values

- How can you add another `key:value` pair to a dictionary?

```
>>> name1
{'age': 18, 'name': 'Sam', 'SN': 4445555}
>>> name1['age']
18
>>> name1['sex'] = 'male'
>>> name1
{'age': 18, 'sex': 'male', 'name': 'Sam', 'SN':
  4445555}
```


Slides courtesy of Dr. Alan McLeod

Dictionaries – Adding Values

- Here’s another option

```
>>> allNames['Boris']['sex'] = 'male'
>>> allNames['Boris']
{'age': 21, 'SN': 5554444, 'sex': 'male'}
```


Slides courtesy of Dr. Alan McLeod

Dictionaries - Cont.

- Dictionary keys must be immutable and unique
  - Don't want to change your key values
  - Don't want duplicate entries
- The dictionary itself is mutable
- You can create an empty dictionary

```
>>> mtDictionary = {}
```

- Add new `key:value` pairs as seen on the previous slides

Slides courtesy of Dr. Alan McLeod

Dictionaries - Cont.

- A dictionary has a method called `keys()` that returns an iterable list of key values

```
>>> name1.keys()
dict_keys(['age', 'sex', 'name', 'SN'])
```

- If you use the `sorted(…)` BIF on a dictionary it returns a dictionary sorted by key
  - See Section 5.5 in the Python Tutorial

Slides courtesy of Dr. Alan McLeod

Elements in Collections

- How do you look through all the elements?
  - Use a loop!
- How do you look for something specific?
  - Loop through all the elements
  - Examine each one – does it satisfy the property or properties you're looking for?

Slides courtesy of Dr. Alan McLeod

Finding Mins, Maxs and Sums

- Naturally Python has BIFs for these!

`min(iter, key=None)`
`min(arg0, arg1, arg2, …, key=None)`,
`sum(iter[, start])`

- **`max`** is called in the same manner as **`min`**
- **`iter`** is a list, tuple or string
- **`key`** is optional and we won't use it
  - Points to a function that determines the order of the elements

Slides courtesy of Dr. Alan McLeod

Finding Mins and Maxs - Cont.

- Sometimes you have to do this yourself
- This function returns the minimum of a simple list

```
def findMin(aList):
    min = aList[0]
    i = 1
    while i < len(aList) :
        if aList[i] < min :
            min = aList[i]
        i = i + 1

    return min
```


Slides courtesy of Dr. Alan McLeod

Finding Mins and Maxs - Cont.

- Maybe you want to know the index of the minimum, not the value
  - You'll need to write your own function
- Using the first element in a collection is a good starting point for the min or max
  - You could start with some very large value for your min or some very small value for your max
  - However, this does not make it easier and you have to know the range of your data or make assumptions
- Demo: FindMinMaxSum.py

Slides courtesy of Dr. Alan McLeod

Finding Mins and Maxs - Cont.

- Note how the functions work with lists of other types
- Note that the built-in `sum(…)` works only with lists of numbers
  - ***We*** can modify ***our*** sum to work with the other list types

Slides courtesy of Dr. Alan McLeod

Finding Mins and Maxs in a Mix

- Suppose your list has a mix of types

```
>>> test = [1, 2, 'abc', 3]
>>> sum(test)
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    sum(test)
TypeError: unsupported operand type(s) for +: 'int'
  and 'str'
>>> min(test)
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    min(test)
TypeError: unorderable types: str() < int()
```


Slides courtesy of Dr. Alan McLeod

Finding Mins and Maxs in a Mix - Cont.

- Hmmmm …
- If we did this ourselves we could use **`isinstance()`** to check types before comparing or summing elements!

Slides courtesy of Dr. Alan McLeod

Timing Code Execution

- We often need to choose code to minimize execution speed, but how can you tell?
  - Sometimes you can predict …
  - Usually you have to measure execution speed in a controlled experiment
- If you are going to compare the speeds of two algorithms, you need to compare them under the same conditions
  - Hardware **and** software

Slides courtesy of Dr. Alan McLeod

Timing Code Execution - Cont.

- Since there is such a variety of hardware and software platforms, all you can conclude from such an experiment is that one algorithm is faster than the other
  - The absolute timing values are not much use

Slides courtesy of Dr. Alan McLeod

Timing Code Execution - Cont.

- To measure timings, import the `time` module and use the **`clock()`** function
  - Get the time before and get the time after
  - Subtract the first from the second for the time elapsed
- Here is what the Python module docs have to say:

> "… this is the function to use for benchmarking Python or timing algorithms.
>
> On Windows, this function returns wall-clock seconds elapsed since the first call to this function, as a floating point number, based on the Win32 function `QueryPerformanceCounter()`. The resolution is typically better than one microsecond."

Slides courtesy of Dr. Alan McLeod

Timing Code Execution - Cont.

- So, just for fun, let's compare our **`findMax(…)`** function to the **`max(…)`** BIF
- We will have to enlarge the list so that it will take enough time to measure
  - We can fill it with random numbers as "fodder"
- Demo: TimingFindMax.py
- Which one is slower? Can we speed up our **`findMax(…)`** function?
  - Note how timings change from one run to the next …

Slides courtesy of Dr. Alan McLeod

Timing Code Execution - Cont.

- Why is the **`for`** loop much faster than the **`while`** loop? (*Good to know!*)
- Why is the BIF still faster than our fastest code?

Slides courtesy of Dr. Alan McLeod