1. Introduction to the Python Programming Environment

J. S. Wright
jswright@tsinghua.edu.cn

1.1 Installing and Using Python

There are several ways to install python. For most users, the easiest option is to install a packaged python distribution, such as Continuum Anaconda or Enthought Canopy.

Anaconda is a free python distribution that provides a number of open source tools for scientific programming using python, including the Spyder integrated development environment (IDE) and the iPython notebook framework. To install Anaconda:

1. Go to `https://www.continuum.io/downloads`.
2. Find the installer for your system. You should install Python 2.7, as these notes and several of the modules we use during the course are specific to Python 2. We will not cover Python 3 (currently 3.5), which is significantly different from Python 2.
3. Follow the instructions to install Anaconda.
4. Package management is done using the `conda` or `pip` commands at the command line. For instructions about using `conda`, see this page. For `pip`, see the documentation here.

You may want to start by installing Iris: type `conda install -c scitools iris` at the command line. This will install a number of climate science-related packages that we will use during the course, and introduce you to how easy it is to use `conda`. The first item (`conda`) runs the package manager. The second item (`install`) tells the package manager that we want to install a package. There are several possible commands; other useful commands include `list`, which lists the installed packages, and `search`, which allows us to look for packages by

name. The third item (`-c scitools` tells the package manager which channel to look in (in this case, it's a channel maintained by the UK Met Office SciTools initiative). The final item is the package we want to install.

Enthought offers academic licenses for Canopy free of charge to staff and students at degree-granting academic institutions, and includes its own built-in IDE. The installation procedure follows:

1. Go to `https://www.enthought.com/products/canopy/`.
2. Click on the button "Get Canopy" and then on the tab "For Academics".
3. Click on "Request your license" and create an account. Be sure to use your academic email address!
4. Once your account is approved, go to the Enthought downloads page and log in to your account.
5. Download the appropriate version of "Canopy with Python Essentials" for your operating system.
6. Follow the instructions to install Enthought Canopy.
7. You will probably want to make Canopy your default python environment.
8. Use the Canopy package manager (Tools > Package Manager) to install new packages and keep your installation up to date.

Note that it is sometimes difficult to connect to the package manager server from within China. If you experience problems, there are other ways to install and update python packages. For example, Canopy package management can also be done via the command line using `pip` (see above).

Linux users can typically install python using the package manager associated with their system, and may wish to look into `pip`, a command line python package manager. The internet contains a multitude of resources that describe alternative methods for installing python and managing python packages, and readers who are not satisfied with the approach described above should consult one of these. Students enrolled in the Atmosphere–Ocean Interactions course are encouraged to contact me if they experience any trouble installing python or using the package manager.

Once you have installed a python distribution, using python is as easy as opening your distribution and loading an IDE window (Spyder for Anaconda or the Canopy editor for Canopy). If you find these IDEs unstable, good alternative IDEs include PyCharm, or simply using a text editor and running python at the command line. The IDE you opened will typically have several panes, including a file browser, an editor, and an IPython shell. You can type python commands directly into the IPython shell, for instance the classic:

```
In [1]:  print 'hello world'
hello world
```

You can also use python as a calculator:

```
In [2]:  1+2
Out[2]:  3
```

Alternatively, you can begin editing a python script by clicking on "Create a new file" and typing in python commands:

```
1 print 'hello world'
2 1+2
```

You can run this code by clicking on the green right-facing triangle at the top of the editor window, selecting "Run File" from the "Run" menu, or using the keyboard shortcut listed to the right of "Run File" in the "Run" menu. You should then get output like:

```
In [3]:  %run myscript.py
hello world
```

Note that no output results from the second line `1+2` when it is contained within a script. This is because the `print` command is required to output data to the screen from inside scripts. We can remedy this by editing the code:

```
1 print 'hello world'
2 print 1+2
```

Running the script now results in

```
In [4]:  %run myscript.py
hello world
3
```

We will learn more about how to use `print` and other python commands as we go along.

If you have chosen not to install Enthought Canopy, then you will probably want to install an integrated development environment (such as IDLE). You can also choose to run python exclusively from the command line (type '`python`' to enter commands interactively and `Ctrl+D` to exit, or run a script by typing '`python myscript.py`'), in which case you will probably want a python-aware editor (such as emacs).

1.2 Data Types

Python contains several intrinsic data types. These include data types that most of you will have already encountered (such as the **integer**, **float**, **string** and **boolean** types), as well as the rather strange `NoneType` type.

Integers and floats are numeric types. The difference, as one might expect, is that integers contain no decimal part, while floats do. Even floats that are equal in magnitude to integers contain the decimal part `.0`, so that `14.0` is the floating point representation of the integer `14`. Assignment of numeric variables in python is straightforward. For example, to set the variable `a` equal to 5, we write

```
In [1]:  a = 5
```

Arithmetic operations on numeric types are also straightforward, and include addition (`a + b`), subtraction (`a - b`), multiplication (`a * b`), division (`a / b`), exponentiation (`a**b`) and the modulo operator (`a % b`), which calculates the remainder from `a / b`. Numeric types can be combined in arithmetic operations, with the output defaulting to the most complex type. For instance, the product of an integer and an integer will be an integer, but the product of an integer and a float will be a float. The most important implication of this is the the quotient of an integer and an integer will also be an integer, so that in python `14/5 = 2` but `14/5.0 = 2.8`.

Python attempts to report all rounding errors honestly. For instance:

```
In [2]:  6.2*2.7
Out[2]:  16.740000000000002
```

As a result, one should be very careful about evaluating equality between two variables. Equality is evaluated as `(a == b)` and inequality as `(a != b)`. Relative comparisons also follow standard conventions (`a > b` for `a` greater than `b`, `a >= b` for `a` greater than or equal to `b`, `a <= b` for `a` less than or equal to `b` and `a < b` for `a` less than `b`). The results of these tests have boolean types.

Boolean variables are binary variables that can take one of two values: `True` or `False`. Boolean variables can be assigned as `a = True` or `b = False`. The capitalization is important in this case, as `False` is a boolean value, but `false` is not. Boolean types can also be included in arithmetic operations: `True` has a numerical value of 1 and `False` has a numerical value of 0. Boolean variables can be combined using the logical and, which is signified by the symbol `&`. The value of `True & True` evaluates to `True`, while the value of `True & False` evaluates to `False`. The logical or is signified by the symbol `|`, where the value of `True | False` evaluates to `True` and the value of `False | False` evaluates to `False`. Logical negation is signified by the symbol ~, where the value of `~True` evaluates to `False` and vice versa.

Strings are sequences of characters, and are assigned by surrounding the contents of the string with either single quotes (`a = 'Tsinghua University'`) or double quotes (`a = ''Tsinghua University''`).

For those who are familiar with compiled languages, such as Fortran or C, `NoneType` variables act like variables that have been declared but not yet defined. For instance, we can create the `NoneType` variable `a` by assignment using `a = None` (the capitalization is again important) to reserve its location in memory prior to assignment. If we then attempt to use that variable before assigning it a non-`NoneType` value, we will receive an error reminding us that `a` does not yet have a value other than `None`.

Python includes a variety of **intrinsic functions** to convert variables from one type to another. Intrinsic functions are functions that are included in the core distribution, and are therefore always accessible. Integers can be converted to floats via the intrinsic function `float()` and floats can be converted to integers via the intrinsic function `int()`, although conversion from floats to integers using `int()` simply discards the decimal part, rather than rounding to the nearest integer:

```
In [3]:  int(12.3)
Out[3]:  12
```

```
In [4]:  int(12.5)
Out[4]:  12

In [5]:  int(12.8)
Out[5]:  12

In [6]:  int(-12.8)
Out[6]:  -12
```

If you want to round a float to the nearest integer, you can do so by first using the intrinsic function `round()`:

```
In [7]:  int(round(12.8))
Out[7]:  13
```

Strings containing valid numeric content can be converted to numeric types via the `int()` or `float()` intrinsic functions, but attempts to convert strings that contain characters will result in an error. Strings containing decimal points cannot be directly converted to integers, but can be first converted to floats and then to integers. Any data type can be converted to a boolean using the intrinsic function `bool()`. Readers should experiment to discover how the results of this conversion vary for different inputs. `NoneType` objects cannot be converted to numeric type objects, but `bool(None)` yields `False` and `str(None)` yields `'None'`.

1.3 Data Structures

In addition to the several intrinsic data types, python also includes several useful intrinsic data structures for storing, organizing and manipulating data. These structures include lists, tuples and dictionaries. Data structures in python have their own types, so lists are of `list` type, tuples are of `tuple` type and dictionaries are of `dict` type. We will often use these structures in our code.

1.3.1 Lists

A list is an ordered set of data. Lists can contain any data type, and in fact the same list can contain multiple data types. We can create a list by putting data into square brackets `[]`, with each value separated by a comma. For example, the list

```
In [1]:  a = [8.314,'December 30, 1984',42,[1,2,3]]
In [2]:  len(a)
Out[2]:  4
```

contains the universal gas constant, the birthdate of LeBron James (as a string), the Answer to the Ultimate Question of Life, the Universe, and Everything, and a list of the integers 1, 2 and 3. The list `a` has a length of 4, which we can discover by typing `len(a)`. We can retrieve the elements of a list using their index, which in python starts from 0:

```
In [3]:  a[0]
Out[3]:  8.314

In [4]:  a[3]
Out[4]:  [1, 2, 3]

In [5]:  a[3][0]
Out[5]:  1
```

If we try to access an index that is larger than the size of the list, we will get an error message:

```
In[6]:  a[4]
---------------------------------------
IndexError Traceback (most recent call last)
<ipython-input-6-54d383a36828> in <module>()
--> 1 a[4]

IndexError:  list index out of range
```

Python allows several different ways of indexing, including negative indices, which wrap around from the end of the list:

```
In [7]:  a[-1]
Out[7]:  [1, 2, 3]
```

We can also **slice** the list to access a range of values:

```
In [8]:  a[0:3]
Out[8]:  [8.314, 'December 30, 1984', 42]

In [9]:  a[:3]
Out[9]:  [8.314, 'December 30, 1984', 42]

In [10]:  a[:-1]
Out[10]:  [8.314, 'December 30, 1984', 42]
```

Each of these slices uses a slightly different syntax, but they all refer to the same elements of the list. The general syntax of a slice is `start:end:stride`, where `stride` can be used to skip elements of the list:

```
In [11]:  a[0:3:2]
Out[11]:  [8.314, 42]
```

It is important to note that the output of a slice includes the value at the index `start` but does not include the value at the index `end`. This differs from many programming languages, and can take some time (and unexpected errors) to get used to. One additional application of slicing is to quickly reverse the order of a list, which we can do by slicing the whole list with a stride of -1:

```
In [12]:  a[::-1]
Out[12]:  [[1, 2, 3], 42, 'December 30, 1984', 8.314]
```

We can also treat strings as lists of characters. For example, if we want to access only the year of LeBron James' birthdate or discover the length of the string, we can type

```
In [13]:  a[1][13:]
Out[13]:  '1984'

In [14]:  len(a[1])
Out[14]:  17
```

This feature makes it quite easy to parse dates, file names and other strings.

Lists are **mutable**, in the sense that we can change their values, add and remove values, and alter the order of the values. For example, we can replace the universal gas constant with the gas constant for dry air by assigning a new value to `a[0]`:

```
In [15]:  a[0] = 287.0
In [16]:  a
Out[16]:  [287.0, 'December 30, 1984', 42, [1, 2, 3]]
```

We can also insert data at a specified location in the list, append data to the end of the list, or remove the first instance of a value from the list:

```
In [17]:  a.insert(1,300)
In [18]:  a
Out[18]:  [287.0, 300, 'December 30, 1984', 42, [1, 2, 3]]

In [19]:  a.append('sixty-four')
In [20]:  a
Out[20]:  [287.0, 300, 'December 30, 1984', 42, [1, 2, 3], 'sixty-four']

In [21]:  a.remove(42)
In [22]:  a
Out[22]:  [287.0, 300, 'December 30, 1984', [1, 2, 3], 'sixty-four']
```

Here, `.insert()`, `.append()`, and `.remove()` are intrinsic methods that are shared by all lists. You should experiment with them for yourself to learn more about out how they behave. Another useful intrinsic method of lists is `.sort()`. For example, suppose we have a list with several integers in some order:

```
In [23]:  x = [19, 3, -9, 67, 32, 5]
```

We can quickly sort this list into ascending order

```
In [24]:  x.sort()
In [25]:  print x
Out[25]:  [-9, 3, 5, 19, 32, 67]
```

The `.sort()` method has several other features that we will neglect for now, but note that `.sort(reverse=True)` will sort the list in descending order rather than ascending order, while `.reverse()` will reverse the order of the list without sorting it (this is similar to `a[::-1]`, but `a.reverse()` changes the order of the list while `a[::-1]` simply returns a view of the list in reversed order without actually changing the list). If you want to see all of the intrinsic methods associated with lists, you can create a list (any list) and type `help(myList)` at the python prompt, where `myList` is the name of your list. This approach will work with any data type, structure or function, not just lists. Even scalar variables have intrinsic functions!

Lists do not need to contain data in order to be lists. For example, we can create an empty list, which has the boolean value `False` and a length of 0:

```
In [26]:  x = []

In [27]:  bool(x)
Out[27]:  False

In [28]:  len(x)
Out[28]:  0
```

All other lists, including `[0]` and `[None]`, have positive lengths and boolean values of `True`. We will encounter situations in which empty lists are useful later in the course.

Python contains a number of intrinsic methods for generating lists. One of the most useful of these is the `range` function, which generates a list of integers given a starting point and an ending point:

```
In [29]:  range(1,10)
Out[29]:  [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Note that `range` excludes the ending point from the resulting list, just like slicing. We can also specify only the ending point, in which case `range` generates a list of integers starting at zero:

```
In [30]:  range(10)
Out[30]:  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

or specify a stride, in which case we must also specify both the starting point and the ending point:

```
In [31]:  range(1,10,2)
Out[31]:  [1, 3, 5, 7, 9]
```

The `range` function will turn out to be quite useful when dealing with `for` loops, as we will see in Section 1.3.3.

If you are familiar with another programming language, it is important to note that while lists look like arrays, lists are not arrays. For instance, consider the sum `a + b` of the lists `a = range(5)` and `b = range(5)`. We might expect the result to be `[0, 2, 4, 6, 8]`, but instead we get:

```
In [32]:  a + b
Out[32]:  [0, 1, 2, 3, 4, 0, 1, 2, 3, 4]
```

The scalar multiplication `a*2` gives a similar result: `a` is repeated twice, rather than the elements of `a` being doubled. We can generate a list in which every element of `a` is doubled using list comprehension (e.g., `b = [i*2 for i in a]`), but this syntax gets very inefficient for more complicated calculations. We will learn how to generate arrays in section 1.5

1.3.2 Tuples

Tuples are also ordered sequences of data. Like lists, tuples can be indexed and sliced, and their lengths can be retrieved using `len()`. Unlike lists, tuples are **immutable**. Once a tuple is defined, we cannot change the values it contains. We define a tuple by placing data between parentheses `()`, with values separated by commas. For example, the tuple `b = (39.9,116.4)` contains the latitude and longitude of Beijing city. Like the actual location of Beijing, these values cannot be changed once they are assigned. For instance, suppose we decided to change the longitude of Beijing so that it is located just south of Chicago:

```
In [33]:  b[1] = 272.3
---------------------------------------
TypeError Traceback (most recent call last)
<ipython-input-33-21aeb9b3e2ce> in <module>()
--> 1 b[1] = 272.3

TypeError:  'tuple' object does not support item assignment
```

We can easily convert tuples to lists and vice versa:

```
In [34]:  list(b)
Out[34]:  [39.9, 116.4]

In [35]:  tuple(a) Out[35]:  (287.0, 300, 'December 30, 1984', [1,
2, 3], 'sixty-four')
```

Note that `list()` and `tuple()` are intrinsic functions that are entirely analogous with `float()` or `str()`. This is a good reason never to name your tuple `tuple` or your list `list`! Tuples have a number of useful applications, some of which we will cover later in these notes.

1.3.3 DICTIONARIES

Dictionaries are unordered data structures, in which each element has a key rather than an index. Dictionaries are defined using the syntax `d = {key0:value0, key1:value1,...}`. The keys are often strings, but can also have numeric (or even boolean) types. Note, however that the keys must be unique, or else the most recently defined value will overwrite any previously defined values. As a result, a dictionary with boolean keys can have at most two values (one for `True` and one for `False`), and is useful only under very specific circumstances (such as if we want to define different sets of behavior when a condition is satisfied or not satisfied).

Dictionaries are most useful when their keys have intuitive meanings. This allows us to create small databases, which we can access without needing to remember exactly what index goes with which property of the data. For example, consider the following script, in which we define a dictionary that stores data that can be used to provide a basic description of presidents of the United States:

```
p = {'name':'Barack Obama',
     'born':1961,
     'inaugurated':'January 20, 2009',
     'number':'44th'
    }

print p['name']+' was born in '+str(p['born'])+\
      ', and became the '+p['number']+\
      ' president of the United States on '+p['inaugurated']
```

We access the values in the dictionary `p` by indexing according to the keys. Note that we must convert the integer value in `p['born']` to a string in order to include it in the output sentence. Running this code gives the output

```
In [36]: %run presidents.py
Barack Obama was born in 1961, and became the 44th president of the
United States on January 20, 2009
```

If we wanted to know what keys are contained in a dictionary, we can use the intrinsic method `p.keys()`, which returns a list containing the keys:

```
In [37]: p.keys()
Out[37]: ['born', 'inaugurated', 'name', 'number']
```

Dictionaries have a number of intrinsic methods. We could list all of these by typing `help(p)`.

We can expand the database by replacing the values in the dictionary `p` with lists:

```
p = {'name':['Millard Fillmore','Barack Obama'],
     'born':[1800,1961],
     'inaugurated':['July 9, 1850','January 20, 2009'],
     'number':['13th','44th']
```

```
    }

for ii in range(len(p['name'])):
    print p['name'][ii]+' was born in '+str(p['born'][ii])+\
      ', and became the '+p['number'][ii]+\
      ' president of the United States on '+p['inaugurated'][ii]
```

which gives the output

```
In [38]: %run presidents.py
Millard Fillmore was born in 1800, and became the 13th president of
the United States on July 9, 1850
Barack Obama was born in 1961, and became the 44th president of the
United States on January 20, 2009
```

This script may not be very useful, but it does show us how to use the `range()` and `len()` intrinsic functions to create a `for` loop that covers the entire database. We could also loop over `p.keys()`, or indeed any list. The contents of the `for` loop are indented. We will see the reasons for this in the next section, and will discuss `for` loops and other (hopefully more practical) uses for dictionaries as we go along.

1.4 Functions and Modules

The following code defines a **function** for calculating the surface area of a sphere, given the radius and a value for the constant π.

```
def SrfAreaSphere(radius,pi=3.14):
    '''
    This function calculates the surface area of a sphere given
    the radius and an approximate value of pi
    '''
    sa = 4*pi*radius**2
    return sa
```

The first line, `def SrfAreaSphere(radius,pi=3.14)`, tells the python interpreter that the following lines are a function called `SrfAreaSphere` that takes the **input** `radius` and the **optional keyword** `pi`. The optional keyword `pi` has the **default value** of `3.14` unless we specifically pass a different value. The next four lines contain a **docstring** that describes the purpose of the function. We should always include a description of this type so that we can better keep track of our code, and we should often include additional information as well (such as the date the function was created or edited, the form of the input data or the definitions of keywords). The information within the triple quotes will a user will see if they load our function and type `help(SrfAreaSphere)`. The next line, `sa = 4*pi*radius**2`, calculates the surface area of the sphere according to the formula $A = \pi r^2$. The last line, `return sa`, returns the calculated surface area.

The first line ends with a colon `:`, indicating that the lines that follow should be treated as a group, or block of code. The last six lines of code (the body of the function `SrfAreaSphere`) are also indented relative to the first. The indent further marks the block of code that belongs to the function: all of these lines are part of its definition. The end of the indented block of code indicates the completion of the function definition. We can find similar indents in other blocks of code, such as conditional `if` statements or `for` loops (see Section 1.3.3). Using indents to organize the code can be confusing for new users, but it makes python code exceptionally easy to read. Many editors (such as the python editors included in Enthought Canopy, IDLE or emacs) will automatically indent the code within a function definition or other code block, but it remains the programmer's responsibility to mark the end of each code block by removing the indent in subsequent lines. Python will also exit with an error if it encounters an indented block of code without an introductory `def`, `if`, `for` or other appropriate statement.

Once we have loaded the function (for instance, by running the script in Canopy), we can use our function to calculate the approximate surface area of the Earth using the mean radius of 6.37×10^6 m:

```
In [2]:  print SrfAreaSphere(6.37e6)
5.09645864e+14
```

The optional keyword `pi` is not required when we call the function `SrfAreaSphere` because we gave it a default value of `3.14` when we defined the function. We could also define a value for the variable `pi` in the function itself (i.e., add another line `pi = 3.14` at line 6). The benefit of including `pi` as an optional keyword is that it allows us to choose to pass a more precise value for π, as in the following example:

```
In [3]:  print SrfAreaSphere(6.37e6,pi=3.14159)
5.09903933084e+14
```

We can also use data structures to pass inputs to the function:

```
In [4]:  args = [6.37e6]
In [5]:  kwds = {'pi':  3.14159}
In [6]:  print SrfAreaSphere(*args,**kwds)
5.09903933084e+14
```

Here, the variable `args` is a list consisting of the arguments to pass to the function as inputs and the variable `kwds` is a dictionary with the keyword names as keys and the values to assign to those keywords as values. This approach is not very efficient in this simple example, but can be quite useful if we need to call a function with many arguments or keywords – especially if we want to call that function many times with the same keywords.

We might also wish to calculate the surface area of the Earth within the script itself. We can do this by adding one line to the script:

```
1 def SrfAreaSphere(radius,pi=3.14):
2     '''
3     This function calculates the surface area of a sphere given
4     the radius and an approximate value of pi
5     '''
6     sa = 4*pi*radius**2
7     return sa
8
9 print SrfAreaSphere(6.37e6,pi=3.14159)
```

Line 9 is not indented, indicating that (1) it is not a part of the function definition and (2) the function definition ends in line 3. The call to `SrfAreaSphere` in line 9 must come after the function definition ends because it references the function. If we run this script within our development environment or from the command line, it will print the value `5.09903933084e+14` to the screen and exit.

The constant π is fairly common, and we may not want to define it every time we need it. A more satisfying solution in python is to **import** the standard **module** `math`, which contains useful mathematical constants (such as π) and functions (such as sin, cos, exp and log). Our modified code could be:

```
1 import math
2
3 def SrfAreaSphere(radius):
4     '''
5     This function calculates the surface area of a sphere given
6     the radius of the sphere
7     '''
8     sa = 4*math.pi*radius**2
9     return sa
```

We have added the line `import math`, removed the optional keyword `pi` from the function definition and replaced the variable `pi` in the formula with the constant `math.pi`. This constant contains a more precise value of π than we have used so far:

```
In [7]:  print math.pi
3.14159265359
```

We must import the `math` module before the function definition if we want to use the constant `math.pi`. There are several ways to import the contents of a module. For instance, if we want to import only the constant `pi`, we could rewrite the code as:

```
1 from math import pi
2
3 def SrfAreaSphere(radius):
4     '''
5     This function calculates the surface area of a sphere given
6     the radius of the sphere
```

```
    '''
    sa = 4*pi*radius**2
    return sa
```

Note that the code inside the function now refers to the constant π as `pi` rather than `math.pi`.

If our code contains several mathematical functions, we could also import the entire contents of the `math` module using the wildcard `*`:

```
from math import *

def SrfAreaSphere(radius):
    '''
    This function calculates the surface area of a sphere given
    the radius of the sphere
    '''
    sa = 4*pi*radius**2
    return sa
```

This approach is generally discouraged, because it can relatively easily to multiply defined functions or constants. For instance, we might (for some reason) also have a variable named `pi` in our code. If we import the entire contents of the `math` module using `from math import *`, then there would be a conflict between our local variable `pi` and the constant `pi` as defined in the module `math` — we may think we are referring to one when we are in fact referring to the other. A better option, which is particularly useful for modules with long names, is to give the imported module a local name:

```
import math as m

def SrfAreaSphere(radius):
    '''
    This function calculates the surface area of a sphere given
    the radius of the sphere
    '''
    sa = 4*m.pi*radius**2
    return sa
```

Our function now references the value of π as `m.pi`. In this case we must be careful not to define any variables with the name `m`, because this would locally overwrite the entire module and any references to `m.pi` would result in errors.

The `math` module is one of thousands of available python modules. Modules are blocks of code containing their own constants, definitions of data types, and intrinsic methods. Importing a module enables local access to whatever portion of the module we import, such as the constant `math.pi`. This structure allows us to keep the code and its execution lean, because we only import modules (or parts of modules) that we intend to use. Importing modules using the `import math` or `import math as m` syntaxes also allows us to avoid variable conflicts among different modules. Such conflicts can be quite common in Fortran or C code.

The remainder of these introductory notes will focus on programming a function that uses the haversine formula to calculate the distance between two points on a sphere:

$$d = r \arccos\left(\sin(\varphi_1)\sin(\varphi_2) + \cos(\varphi_1)\cos(\varphi_2)\cos(\lambda_2 - \lambda_1)\right)$$

where d is distance, r is the radius of the sphere, φ_1 and φ_2 are the latitudes of the two points, and λ_1 and λ_2 are the longitudes of the two points.

Based on what we have seen so far, we might construct a first guess that uses familiar data structures and the `math` module:

```
import math

def Distance(xy1,xy2,r=6.371e3):
    '''
    This function uses the haversine formula to calculate
    distance on a sphere with radius r

    Inputs:
      xy1: a tuple containing the first (lon,lat) pair
      xy2: a tuple containing the second (lon,lat) pair
      r: the radius of the sphere; defaults to Earth

    Outputs:
      distance between xy1 and xy2 in same units as r
    '''
    d2r = math.pi/180. # factor to convert deg to radians
    lm1 = xy1[0]*d2r
    ph1 = xy1[1]*d2r
    lm2 = xy2[0]*d2r
    ph2 = xy2[1]*d2r
    # calculate distance
    hvs = math.sin(ph1)*math.sin(ph2) + \
        math.cos(ph1)*math.cos(ph2)*math.cos(lm2-lm1)
    dst = r*math.acos(hvs)
    return dst
```

We can use this code to calculate the distance between two points, such as Beijing and Chicago:

```
In [2]:  Distance((116.4,39.9),(272.3,41.8))
Out[2]:  10610.539645205401
```

Note that the trigonometric functions in the `math` module expect angles in radians rather than degrees, so we must first convert the latitude and longitude of each point to radians. Note also that we could just as easily define four inputs `x1`, `y1`, `x2` and `y2` rather than the two tuples `xy1` and `xy2`; the code would be effectively the same.

Our code is fine for two individual points, but what if we want to calculate the distance between one point and a group of other points? Perhaps we could try a list of tuples, for which we might change the code to something like:

```
import math

def Distance(xy1,xy2,r=6.371e3):
    '''
    This function uses the haversine formula to calculate
    distance on a sphere with radius r

    Inputs:
      xy1: a tuple containing the first (lon,lat) pair
      xy2: a list containing (lon,lat) tuples
      r: the radius of the sphere; defaults to Earth

    Outputs:
      distance between xy1 and xy2 in same units as r
    '''
    d2r = math.pi/180. # factor to convert deg to radians
    lm1 = xy1[0]*d2r
    ph1 = xy1[1]*d2r
    dst = []
    for ii in range(len(xy2)):
        lm2 = xy2[ii][0]*d2r
        ph2 = xy2[ii][1]*d2r
    #-- calculate distance for this point
        hvs = math.sin(ph1)*math.sin(ph2) + \
            math.cos(ph1)*math.cos(ph2)*math.cos(lm2-lm1)
        dst.append(r*math.acos(hvs))
    return dst
```

Feeding in the locations for Beijing as the tuple `xy1` and the locations for Chicago, Moscow, Wellington, Khartoum and Sao Paulo as the list of tuples `xy2` gives the result:

```
In [3]:  xy1 = (116.4,39.9)
In [4]:  xy2 = [(272.3,41.8),(37.6,55.75),(174.8,-41.3),(32.5,15.6),
(313.4,-23.5)]
In [5]:  Distance(xy1,xy2)
Out[5]:  [10610.539645205401, 5794.919107386599, 10782.650595522533,
8391.021179698164, 17592.529455169093]
```

The program works, but the format of the output is not very satisfying. In particular, it is difficult to match the distances to their respective locations. Another option would be to replace the list with a dictionary, using the names of the cities as keys. The `Distance` program also needs to be changed, to something like:

```
import math

def Distance(xy1,xy2,r=6.371e3):
    '''
    This function uses the haversine formula to calculate
    distance on a sphere with radius r
```

```
7
8       Inputs:
9         xy1: a tuple containing the first (lon,lat) pair
10        xy2: a dictionary containing (lon,lat) tuples
11        r: the radius of the sphere; defaults to Earth
12
13      Outputs:
14        distance between xy1 and xy2 in same units as r
15      '''
16      d2r = math.pi/180. # factor to convert deg to radians
17      lm1 = xy1[0]*d2r
18      ph1 = xy1[1]*d2r
19      dst = {}
20      for kk in xy2.keys():
21          lm2 = xy2[kk][0]*d2r
22          ph2 = xy2[kk][1]*d2r
23      #-- calculate distance for this point
24          hvs = math.sin(ph1)*math.sin(ph2) + \
25              math.cos(ph1)*math.cos(ph2)*math.cos(lm2-lm1)
26          dst[kk] = r*math.acos(hvs)
27      return dst
```

The loop is now over the keys for the dictionary xy2, rather than over the indices of the list xy2. We can populate the output dictionary dst by creating an empty dictionary and adding each key from xy2 as we come to it. Feeding in the same data as before, we get the result:

```
In [6]:  xy2 = {'Chicago':(272.3,41.8),
...   'Moscow':(37.6,55.75),
...   'Wellington':(174.8,-41.3),
...   'Khartoum':(32.5,15.6),
...   'Sao Paulo':(313.4,-23.5)}
In [7]:  Distance(xy1,xy2)
Out[7]:
{'Chicago':  10610.539645205401,
'Khartoum':  8391.021179698164,
'Moscow':  5794.919107386599,
'Sao Paulo':  17592.529455169093,
'Wellington':  10782.650595522533}
```

Note that the order of the keys in the output dictionary is different from the order of the keys in the input dictionary. This is a great reminder that dictionaries are not ordered, and we should not expect them to be.

This solution works well for a limited number of points, but becomes inefficient for larger numbers of points. For instance, suppose we want to use some satellite observations to characterize the atmosphere near Beijing for a specific day. We know what locations the satellite observed throughout the day, but we don't know when the satellite passed close to Beijing. We want to calculate the distance between the satellite observation and Beijing throughout the entire day and find the closest point or points. This calculation may involve

calculating many thousands of distances. Although we could do it with the standard data structures, it would be very inefficient. To do it well, we need better tools.

1.5 NUMPY

The `numpy` module is critical for scientific computing in python. This module is called `numpy`, and it supersedes the `math` module that we encountered in the previous section. Among other things, the `numpy` module is important because it provides an array data structure. We will only introduce a small part of the full functionality of `numpy` in this course, and only the bare minimum of that in this section.

First, let's look at the code using `numpy` instead of `math`:

```
import numpy as np

def Distance(xy1,x2,y2,r=6.371e3):
    '''
    This function uses the haversine formula to calculate
    distance on a sphere with radius r

    Inputs:
      xy1: a (lon,lat) tuple for the base point
      x2: a numpy array (or list) of longitudes
      y2: a numpy array (or list) of latitudes
      r: the radius of the sphere; defaults to Earth

    Outputs:
      array of distances to xy1 in same units as r
    '''
    d2r = np.pi/180. # factor to convert deg to radians
    lm1 = xy1[0]*d2r
    ph1 = xy1[1]*d2r
    lm2 = d2r*np.array(x2)
    ph2 = d2r*np.array(y2)
    #-- calculate distance for the entire array at once
    hvs = np.sin(ph1)*np.sin(ph2) + \
          np.cos(ph1)*np.cos(ph2)*np.cos(lm2-lm1)
    dst = r*np.arccos(hvs)
    return dst
```

We have imported the `numpy` module using the syntax `import numpy as np`. This is the most commonly used syntax for importing `numpy`, and I highly recommend that you use it in your code.

Several differences are immediately apparent. First, we have replaced the constant `math.pi` and the trigonometric functions from `math` with equivalent versions from `numpy` (such as the constant `np.pi`). We will find that `numpy` includes all of the constants and methods that we could get from `math` (although some of the names may differ, such as `math.acos()` and `numpy.arccos`), along with a great many useful methods that `math` does not include. For this reason, we will very rarely use the `math` module for scientific programming. Second, the

docstring indicates that the inputs x2 and y2 can be either lists or numpy arrays. We ensure that they are numpy arrays in the code by using the conversion method `np.array(x2)`, where x2 would most likely be a list, a tuple, or another array (there is no harm in converting an array to an array). It typically does not make sense to turn dictionaries or scalars into numpy arrays, although dictionaries of numpy arrays can be quite useful. Third, unlike when we used lists, tuples or dictionaries, we can multiply the entire array by the constant `d2r` at the same time. Similarly, we can apply `np.sin`, `np.cos` or `np.arccos` to the entire array at once. This means we can remove the loop from the function. This is important once we start working with very large data structures, because here we are using python as an interpreted language (like Matlab, IDL, or NCL) rather than a compiled language (like Fortran or C). Loops are very efficient in compiled languages, but very inefficient in interpreted languages. The fact that this code does not contain loops doesn't mean that there are no loops; however, unlike before, the loops are hidden in the numpy code (which is written in Fortran and C and has already been compiled). As a result, using numpy arrays is typically much faster than using lists.

There are a number of alternative possibilities for this function. For instance, instead of passing two n-element arrays x2 and y2, we could pass one $2 \times n$-element array that contains both the latitudes and longitudes. We could calculate the distance between n pairs of points (as opposed to the distance between one point and n other points) by defining the input variable xy1 as another $2 \times n$ array. We could also define xy1 as a $2 \times m$-element array and return an $m \times n$ array of distances. Using `numpy` makes all of these possible tasks much easier and more efficient.

To use the function for the purpose mentioned at the end of Section 1.4, we need to import another module to read in the satellite data.

```
import numpy as np
import h5py

def Distance(xy1,x2,y2,r=6.371e3):
    '''
    This function uses the haversine formula to calculate
    distance on a sphere with radius r

    Inputs:
      xy1: a (lon,lat) tuple for the base point
      x2: a numpy array (or list) of longitudes
      y2: a numpy array (or list) of latitudes
      r: the radius of the sphere; defaults to Earth

    Outputs:
      array of distances to xy1 in same units as r
    '''
    d2r = np.pi/180. # factor to convert deg to radians
    lm1 = xy1[0]*d2r
    ph1 = xy1[1]*d2r
    lm2 = d2r*np.array(x2)
    ph2 = d2r*np.array(y2)
    #-- calculate distance for the entire array at once
    hvs = np.sin(ph1)*np.sin(ph2) + \
          np.cos(ph1)*np.cos(ph2)*np.cos(lm2-lm1)
```

```
    dst = r*np.arccos(hvs)
    return dst

# specify the directory containing the satellite data
ddir = '/Users/jswright/projects/Aerosols/ConvectiveTransport/data/
    case/'
# open the data file
fco  = h5py.File(ddir+'MLS-Aura_L2GP-CO_v03-30-c01_2007d021.he5','r'
    )
# read in the x-y-time positions of the data
mlx  = fco['HDFEOS/SWATHS/CO/Geolocation Fields/Longitude'][:]
mly  = fco['HDFEOS/SWATHS/CO/Geolocation Fields/Latitude'][:]
mlt  = fco['HDFEOS/SWATHS/CO/Geolocation Fields/Time'][:]
# close the file
fco.close()
# convert time from seconds since January 1, 1970 into hours since
    beginning of day
mlt = (mlt - mlt[0])/3600.
# location of Beijing
xy1 = (116.4, 39.9)
# calculate distance between Beijing and satellite position
dst = Distance(xy1,mlx,mly)
```

We will not discuss the h5py module, except to note that it enables us to read data from files in HDF5 or HDFEOS format. Running this script yields an array of distances with 3484 elements, ranging in magnitude from approximately 111 km to almost 20000 km:

```
In [2]:  dst.shape
Out[2]:  (3484,)

In [3]:  print dst.min()
110.90623

In [4]:  print dst.max()
19767.785
```

It is worth noting that the intrinsic function `len()` can be misleading for numpy arrays, and its use should be replaced by `.shape` (an intrinsic feature of numpy arrays).

1.6 Basic Plotting

After running the previous script, we can use the `matplotlib.pyplot` module to plot the evolution of distance during the day. If we add the following lines to the previous script

```
import matplotlib.pyplot as plt

fig = plt.figure()
axs = fig.add_subplot(1,1,1)
axs.plot(mlt,dst,color='b',linestyle='-',linewidth=2)
plt.show()
```

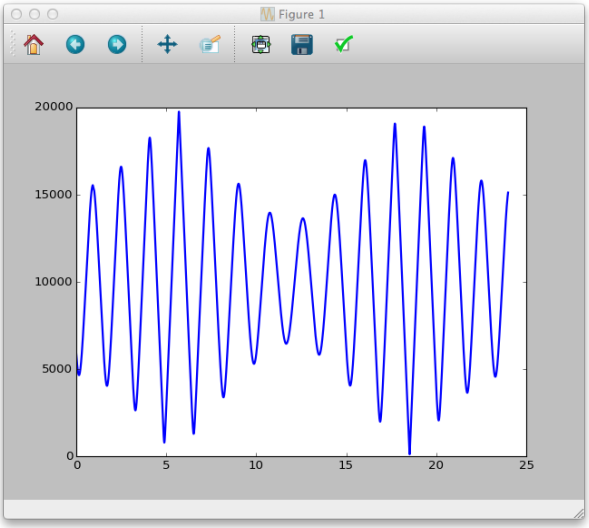

Figure 1.1: Matplotlib plot of distance between Beijing and the Aura Microwave Limb Sounder satellite footprint as a function of time on January 21, 2007, generated using `matplotlib.pyplot.show()`.

and then run the script in its entirety, Figure 1.1 pops up.

Without getting into the many features available through the `matplotlib.pyplot` module, we can refine the plot parameters and save the figure directly by changing the figure generation code to

```
import matplotlib.pyplot as plt

fdir = '/Users/jswright/courses/AtmosphereOceanInteractions/2015/
    notes/python/lecture01/figs/'
fig = plt.figure()
axs = fig.add_subplot(1,1,1)
axs.plot(mlt,dst,color='b',linestyle='-',linewidth=2)
axs.set_title('Distance to Beijing')
axs.set_xlim((0,24)) # x-axis goes from 0 to 24
axs.set_xticks(range(0,25,6)) # ticks every four hours
axs.set_xlabel('Time')
axs.set_ylabel('Distance [km]')
plt.savefig(fdir+'distance_to_beijing.png')
```

Here, we have modified the code to specify the range of the x-axis and the locations of the x ticks, put labels on the axes, and add a title to the plot (see Fig. 1.2). Extensive documentation for `matplotlib` is available online, along with a large gallery of example plots.

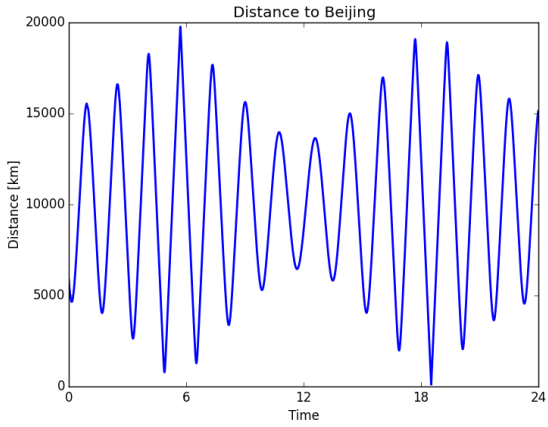

Figure 1.2: Matplotlib plot of distance between Beijing and the Aura Microwave Limb Sounder satellite footprint as a function of time on January 21, 2007, generated using `matplotlib.pyplot.savefig()`.