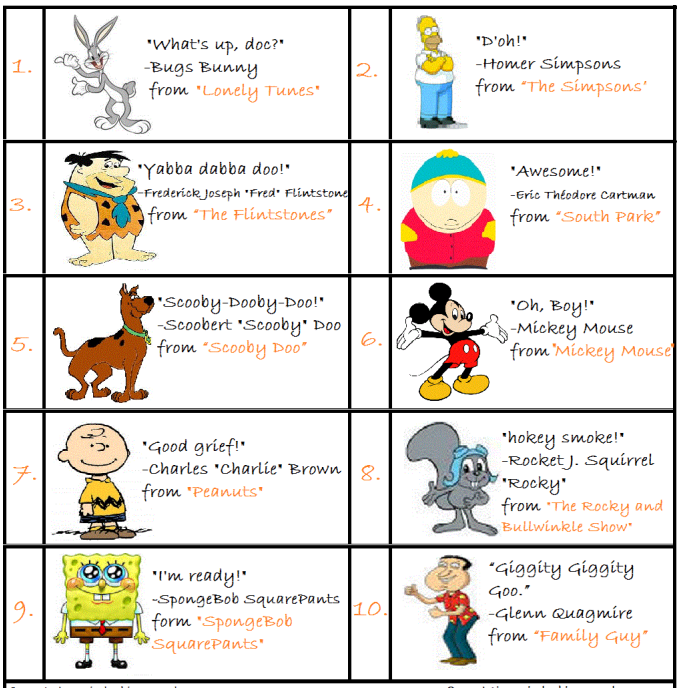
Lists of Data
Mathematical Programming with Python

MATH 2604: Advanced Scientific Computing 4
Spring 2025
Monday/Wednesday/Friday, 1:00-1:50pm

https://people.sc.fsu.edu/~jburkardt/classes/python_2025/lists/lists.pdf

---



A list is an indexed group of data

- *A Python list allows us to refer to several items as a group;*
- *A list looks like items separated by commas and surrounded by square brackets;*
- *Items can be of any datatype, and can be of different data types;*
- *Lists are "mutable"; we can alter, add or delete elements;*
- *A* `for()` *statement can access each list item in turn: a list is "iterable";*
- *Methods allow us to add, delete, count, sort elements;*
- *We can read a text file into a list of sentences or words;*
- *A numeric matrix can be created as a list; but we will much rather do this with the numpy() library!*

1 A list is a list...

A Python list is an object containing elements in a particular order. (Unlike a mathematical set, order matters and elements can be repeated.) The easiest way to create a list is by specifying the elements in order, separated by commas, inside square brackets:

```
grades = [ 75, 80, 88, 75, 81 ]
```

or, alternatively, the `list()` function can be used:

```
grades = list ( ( 75, 80, 88, 75, 81 ) )
```

We reference any single element by a zero-based indexing scheme:

```
grades[1]
```

or a sequence of elements using a pair of values. As with the `range()` function, we don't include the final upper limit value.

```
grades[0:3]
```

Character strings can be elements of a list:

```
months = [ 'January', 'February', 'March', ..., 'December' ]
```

The elements of a list can be a variety of types, including another list:

```
variety = [ 1, 'Hello', [4, 5], 7.6 ]
```

Note that element 2 of this list is the list [4,5], and element 3 is the number 7.6.

Similarly to how we get the length of a character string, we can request the length of a list (the number of elements) with the `len()` function:

```
len ( 'Hello' )
len ( grades )
len ( months )
```

Is the length of the `variety` list equal to 4 or 5?

Suppose you know that element 2 of the `variety` list is also a list. How could you determine the length of that internal list? How could you access the second component of element 2?

2 Let's make a list!

To create a Python list, we can simply list our elements, separated by commas, and enclosed by square brackets:

```
colleges = [ 'Carlow', 'Chatham', 'CMU', 'Duquesne', 'University of Pittsburgh' ]
print ( colleges )
print ( colleges[1] )    # Returns an element
print ( colleges[1:4] )  # Returns a list!
print ( colleges[-1] )
print ( "  Number of colleges in list is ", len ( colleges ) )  # len(list) gives length.
```

There is also a **list()** function which takes its arguments and packs them into a list:

```
colleges = list ( 'Carlow', 'Chatham', 'CMU', 'Duquesne', 'University of Pittsburgh' )
```

It seems silly to have two ways to make a list this way; however we will see later that the **list()** function can also be useful in converting some Python objects into lists, in particular, the output of the **range()** function.

We have mentioned that the `list()` function can capture all the output of `range()`. So if we want a list of even numbers from 10 to 20, we could write:

```
values = list ( range ( 10, 21, 2 ) )
```

3 Reading text into a list

In the Spelling-Bee example, we created a list by reading words from a file. While we still don't completely understand this command, we can see where the list is created

```
with open ( filename ) as fi:
  words = [ word.strip().upper() for word in fi ]
```

Just for fun, let's take the small file *grook.txt* and

```
with open ( 'grook.txt' ) as fi:
  words = [ word.upper() for word in fi ]
print ( words )
```

and then add `strip()` to get rid of the
`n` characters:

```
with open ( 'grook.txt' ) as fi:
  words = [ word.strip().upper() for word in fi ]
print ( words )
```

and then just call `split()` to get individual words:

```
with open ( 'grook.txt' ) as fi:
  words = [ word.split() for word in fi ]
print ( words )
```

4 Modifying a list

We can modify a list by replacing or changing a value specified by its index, or by appending a new value (it goes at the end), or removing a particular value.

```
colleges[3] = 'Robert Morris'
print ( colleges )
```

We can add another element to the end of the list with `append()`, or use `insert()` to insert it into a given position.

```
colleges.append ( 'Point Park' )
print ( colleges )
colleges.insert ( 2, 'St Vincent' }
print ( colleges )
```

We can make the list shorter by using `remove()` to remove a single instance of a given value, or `del()` to remove the value at a given index:

```
colleges.remove ( 'Chatham' )
print ( colleges )
del colleges[0]
print ( colleges )
```

We can also use the `list.pop()` command to extract an item by index, removing it from the list.

```
print ( colleges )
value = colleges.pop ( 1 )
print ( '  value = colleges.pop(1) = ', value )
print ( colleges )
value = colleges.pop ( 0 )
print ( '  value = colleges.pop(0) = ', value )
print ( colleges )
```

If you don't specify an index to `pop()` it takes the last element.

```
  value = colleges.pop ( )
  print ( '  value = colleges.pop() = ', value )
  print ( colleges )
```

so that the `pop()` command makes it easy to implement a classic data structure known as a *stack*.

5 A for() loop can range over a list

When we considered `for()` statements, we hadn't really been officially introduced to lists, but we had an example in which the values to be selected were not generated using the `range()` function, but instead were simply specified in a list. Since the value of US coins is a bit irregular, it is easiest to simply list them:

```
UScoins = [ 1, 5, 10, 25, 50, 100 ]
sum = 0
for coin in UScoins:
  sum = sum + coin
print ( '  The sum of a collection of US coins is ', sum )
```

Instead of numeric data, we might want to cycle through a sequence of string values:

```
friends = [ 'Alice', 'Bob', 'Carol', 'David' ]
for friend in friends:
  print ( '  My friend ', friend, ' has a name of length ', len ( friend ) )
```

We can also cycle through a set of pairs of values, using a list of lists. To select pairs (n, k) for the combinatorial function, we might write:

```
from math import comb
for n, k in [ [ 5, 2 ], [ 10, 3 ], [ 15, 7 ] ]:
  print ( comb ( n, k ) )
```

6 Lists of mixed types

The items in a list don't have to have a common type. A single list could include name (a string), weight (a real number), height in inches (an integer), and is-vacccinated (logical, with value True or False):

```
  patient0 = [ 'Robert Baratheon', 235.4, 73, False ]
  patient1 = [ 'Arya Stark',       134.7, 51, True ]
  patient2 = [ 'Brienne Tarth',    150,   68, True ]
```

Assuming the third item is height, we ask whether Arya Stark is shorter than Brienne Tarth:

```
  arya_shorter = patient1[2] < patient2[2]
```

7 A list of lists

In the example of a doctor's patient registry, each item `patient0`, `patient1` and `patient2` is a separate list. The doctor might want a single master list of all the patients, while retaining the ability to call up the record for any particular patient. It is easy to do this, although now it's important to realize that each element of data is stored inside of a list inside of another list!

```
  Patients = [ patient0, patient1, patient2 ]
```

What do we get if we ask for `len(Patients)`? Now, `Patients[1]` is the entire record for Arya Stark.

```
print ( Patients[1] )
```

To request just her height, we have to specify two indices, something like the row and column:

```
arya_height = Patients[1][2]
```

To see this a little more clearly, here is how Python sees the `Patients` list:

```
  # Col 0                Col 1   Col 2   Col 3
Patients = [
  [ 'Robert Baratheon', 235.4, 73,     False ]   # <-- Row 0
  [ 'Arya Stark',       134.7, 51,     True ]    # <-- Row 1
  [ 'Brienne Tarth',    150,   68,     True ]    # <-- Row 2
]
```

And in fact, this statement (without the row and column comments) could be used to enter the `Patients` list as a single command.

Suppose a new patient wishes to be treated by the doctor. The receptionist creates a new list containing the information for this patient:

```
patient3 = [ 'Tyrion Lannister', 100.5, 38, True ]
```

Now it's necessary to add this patient's records to the single object `Patients`, which contains all the information. We do this using the `.append()` method:

```
Patients.append ( patient3 )
```

To verify that this worked correctly, we can type the entire list

```
print ( Patients )
```

or just check the single new record:

```
print ( Patients[3] )
```

Robert Baratheon dies. How do we get rid of his record in the `patients` list? (Presumably, the `del()` function will be easier to work with than `remove()`.

You can use a `for()` loop to cycle through all the patient records, as long as each row has the same kind of information. Now each time the `for()` statement is executed, the values in the next list are extracted. The `for()` loop needs to supply temporary names for these items, which can then be used within the body of the loop. Let's return to `Patients`, our list of patient data:

```
for name, weight, height, vax in Patients:
  print ( name, 'weighs ', weight, 'pounds, vax status is ', vax )
  if ( not vax ):
    print ( 'Hey', name, 'get vaccinated!' )
```

8 Am I on the list?

You can find out whether an item `x` is already an element of a list `mylist` by using the **in** operator:

```
variety = [ 1, 'Hello', [4, 5], 7.6 ]
'Hello' in variety  # True
[4,5] in variety    # True
98 in variety       # False
```

This is essentially the same `in` operator that we use as part of the `for()` statement. By the way, to ask if something is *not* in the list you use the `not in` expression:

```
  variety = [ 1, 'Hello', [4, 5], 7.6 ]
  'Hello' not in variety      # False
  'Goodbye' not in variety    # True
```

9 The sorted() function

The items in a list are given in the order that was specified when the list was created. However, it is possible to sort them, using the `sorted()` function:

```
  primes = [ 2, 31, 5, 61, 17, 3 ]
  sorted_primes = sorted ( primes )
```

Things get more complicated in the realistic setting where we have a list of lists. In that case, you need to specify a `key`. We will also need to use the `lambda` function, which we won't explain yet!

```
grades = [ ( 'john', 'A', 15 ), ('sam', 'D', 6 ), ('jane', 'B', 12 ) ]

result = sorted ( grades, key = lambda record: record[2] )
print ( result )
[('sam', 'D', 6), ('jane', 'B', 12), ('john', 'A', 15)]

result = sorted ( grades, key = lambda record: record[1] )
print ( result )
[('john', 'A', 15), ('jane', 'B', 12), ('sam', 'D', 6)]

result = sorted ( grades, key = lambda record: record[0] )
print ( result )
[('jane', 'B', 12), ('john', 'A', 15), ('sam', 'D', 6)]
```

10 Creating a list of primes

To see a computational example in which a list might be useful, let's suppose that we want to create a list `prime` of prime numbers less than 1000. Presumably, our list starts out empty. We can use a `for()` loop to run through the values of n from 1 to 1000. We can call `isprime()` from `sympy`, or use one of the prime checkers we wrote ourselves. Each time we find that the value n is prime, we have to add it to the end of our list. At the end, we should announce how many primes we found, and print the list.

```
prime = []
for n in range ( 1, 1000 ):
  is n prime?
  if n is prime
    add n to the list       # How do we do this?
print number of primes      # How do we know this?
print list of primes
```

If 37 is in the list of primes, then the following expression is True:

```
  37 in prime
```

Similarly, for the "friends" example above, the following expression is False:

```
  'Elmo' in friends
```

If we are very forgetful, and we don't know whether Elmo is already in our friends list, we might say:

```
if ( 'Elmo' not in friends ):
  friends.append ( 'Elmo' )
```

and if we suddenly have a fight with Alice, you should know that you could remove her from your friends list.

11 Careful when copying!

In Python, the equals sign (which we think of as the assignment operator) doesn't always work the way you would expect. Python thinks of the name of a list as "pointing" to the list. So when we write `A = [ 1, 2, 3]`, Python creates an object with the appropriate values, and then says, essentially, "If you ever want to see these values again, ask for **A**." Now suppose we issue the Python command `B = A`. Python says, "I see you want to make another name to refer to the same data. OK, either **A** or **B** will work the same now." What happens next may not be what you expect!

```
odds = [ 1, 3, 5, 7, 9 ]
prime = odds              # This does NOT make a new set of data, just a new pointer
prime.remove ( 9 )
prime.remove ( 1 )
print ( 'prime = ', prime )
print ( 'odds = ', odds )
```

Luckily, we can avoid this problem by using the `list.copy()` method, which guarantees that Python creates a new pointer **and** a new set of data:

```
odds = [ 1, 3, 5, 7, 9 ]
prime = odds.copy()             # This makes a new pointer and new set of data
prime.remove ( 9 )
prime.remove ( 1 )
print ( 'prime = ', prime )
print ( 'odds = ', odds )
```

This feature of Python is there for a good reason, but it sometimes contradicts the way a programmer thinks, and so will occasionally give you a moment or two of confusion!

12 Creating lists by reading files

One of the primary uses of lists is to handle data. Often that data is stored in a text file. In a simple case, each line of the file is a record, containing values for a fixed number of data fields. In other cases, the file might simply be a document or a book. To Python, a text file of n lines can be regarded as a list of n entries. Each line of a text file is terminated by an invisible "new line" character, which is sometimes symbolized by \n. Consider the following tiny file, a poem by Piet Hein, stored in *grook.txt*:

```
The road to wisdom?  Well, it is plain
And simple to express:
Err and err and err again,
But less and less and less.
```

If we use the Python function `readlines()`, we get the whole file read into a data structure:

```
input = open ( 'grook.txt', 'r' )
text = input.readlines ( )
print ( text )
['The road to wisdom?  Well, it is plain\n',
 'And simple to express:\n',
 'Err and err and err again,\n',
 'But less and less and less.\n']
```

As you can see, our document has become a list of four strings, each terminated by a newline character. For instance, `text[2]` is the string *'Err and err and err again, \n'*

If we simply want to eliminate the newline character from our results, we can read the file one line at a time, and apply the `strip()` method to each line.

```
input = open ( 'grook.txt', 'r' )
text = []
for line in input:
  text.append ( line.strip ( ) )
print ( text )
['The road to wisdom?  Well, it is plain',
 'And simple to express:',
 'Err and err and err again,',
 'But less and less and less.']
```

Often, we need to look at the individual words in each line of our file. In that case, the `split()` method will help us:

```
input = open ( 'grook.txt', 'r' )
text = []
for line in input:
  text.append ( line.split ( ) )
print ( text )
```

Now we see our text split into lines and individual words. Note, however, that instead of a list of strings, we now have a list of lists of words.

```
[ ['The', 'road', 'to', 'wisdom?', 'Well,', 'it', 'is', 'plain'],
  ['And', 'simple', 'to', 'express:'],
  ['Err', 'and', 'err', 'and', 'err', 'again,'],
  ['But', 'less', 'and', 'less', 'and', 'less.']
```

13 Getting help:

If you have questions about `strip()` or `split()` or the many other methods available for character strings, you can use the command:

```
  help ( str )
```

which is the way to find out the names and meanings of all the methods available for application to objects of the class `str`, that is, "strings".

14 A homework example:

For an upcoming homework exercise, the text file consists of a list of five letter words; each word is on its own line, so using `split()` will introduce an extra set of square brackets around each word, that will make your work more difficult!

```
words = []
input = 'five_letters.txt'
for line in input:
  words.append ( line.split() )
print ( words )
```

results in

```
  [ ['which'], ['there'], ['their]', ... ['pupal'] ]
```

Instead, we can invoke the `strip()` function which will give you a simple list:

```
words = []
input = 'five_letters.txt'
for line in input:
  words.append ( line.strip() )
print ( words )
```

with the result:

```
  [ 'which', 'there', 'their', ... 'pupal' ]
```

15 A list of lists as a matrix

A list can contain items of any data type, and a list of lists can have some rows longer than others. In contrast, a mathematical matrix is typically an $m \times n$ array whose entries are all numbers, and usually real numbers at that.

If we know the dimensions of the matrix in advance, then it's not too hard to set it up. Here is a small version of a matrix often used to approximate the second derivative:

```
  D = [
    [ -2,  1,  0,  0,  0 ],
    [  1, -2,  1,  0,  0 ],
    [  0,  1, -2,  1,  0 ],
    [  0,  0,  1, -2,  1 ],
    [  0,  0,  0,  1, -2 ] ]
```

And if v is a vector of length 5, we can compute the matrix-vector product $u = D * v$ by summing up terms using two `for()` loops to repeatedly execute the operation

```
  u[i] = u[i] + D[i][j] * v[j]
```

In more complicated situations, where we don't know the size of the matrix in advance, simply setting up space for the matrix is complicated, but possible.

So we can use a list of lists to store the entries of a mathematical matrix. However, Python won't really be able to help us do many of the mathematical or linear algebraic operations we might want to perform, such as transpose, determinant, matrix-vector multiplication, solution of a linear system, and so on. We are free to code such algorithms up ourselves, but this is a somewhat painful process. We will look at an example of this idea next time, and soon we will see a whole new library that will make matrix and vector calculations much easier to program.