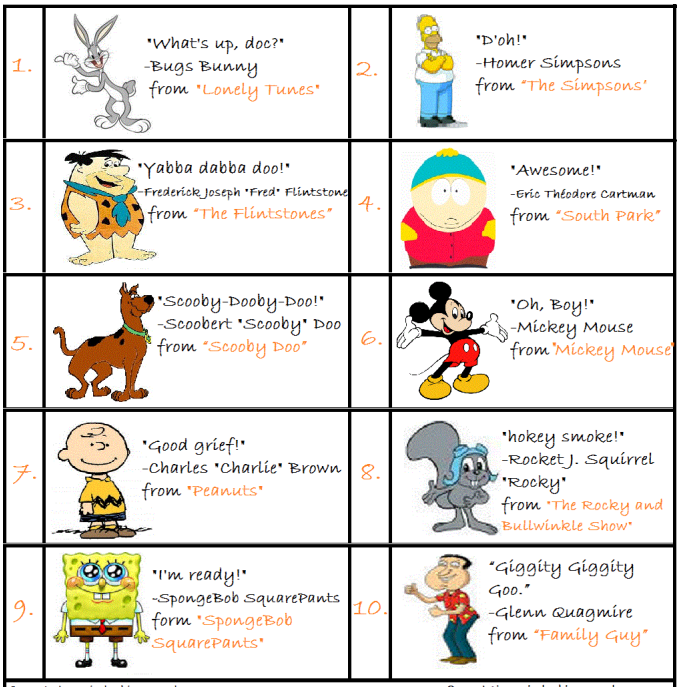
Lists of Data

https://people.sc.fsu.edu/~jburkardt/classes/math1800_2023/python_list/python_list.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 any element;*
- *A* `for()` *statement can access each list item in turnl a list is* **iterable***;*
- *Methods allow us to add, delete, count specific elements;*

1 A list is a list...

A Python list allows us to create an object containing elements in a particular order. 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 ) )
```

Once we have creaed a list, we can reference any single element by the usual zero-based indexing scheme, or a sequence of elements using a pair of values:

```
grades[1]
grades[0:3]
```

Character strings can also be elements of a list:

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

We can request the length of a list (the number of elements) with the `len()` function:

```
len ( grades )
len ( months )
```

2 Let's make a list!

To create a Python list, we literally 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.
```

3 Modifying a list

We can modify a list by replacing or modifying an indexed value, or 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 inserts 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*.

4 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]
```

5 Iterating on 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 ) )
```

You can even do this for a list of lists, as long as each row has the same kind of information:

```
for name, weight, height, vax in patients:
  print ( name, 'weighs ', weight, 'pounds, vax status is ', vax )
```

6 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.

7 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!

8 A list of lists

In the example of a doctor's patient registry, each item `patient0`, `patient1` and `patient2` is a 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 two nested lists:

```
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 simply type

```
print ( patients )
```

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()`.

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

Now we see our text split into lines and individual words. Note, however, that this result differs from the previous one in two ways. Instead of a list of strings, we now have a list of lists of words.

10 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".

11 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 `strip()` will give you a simple list, whereas `split()` will introduce an extra set of square brackets that will make your work more difficult!

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

12 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.