

SEE1002

Introduction to Computing for Energy and  
Environment

Part 2: Elements of Python programming

**Sec. 2: Elementary data structures**

# Course Outline

## **Part 1: Introduction to computing**

## **Part 2: Elements of Python programming**

Section 1: Data and variables

Section 2: Elementary data structures

Section 3: Branching

Section 4: Flow control

## **Part 3: Basic Python programming**

Section 1: Structure of a Python program

Section 2: Input and output

Section 3: Modules

Section 4: Good programming practices

## **Part 4: Python for science and engineering**

Section 1: Vectors, matrices and arrays

Section 2: NumPy and SciPy

# I. What is a data structure?

# Operations on variables

we've already seen this!

There are many, many things that we can do to variables or data. For simplicity, we can classify them as follows:

1. Data manipulation (numeric and string operators)
2. Data comparison (relational and logical operators)
3. Data input/output (`print`, `input`)
4. *Data consolidation or organisation*

# What do we mean by data consolidation?

- For simple programs, each piece of data can have its own dedicated variable.
  - ▶ area, length, width
  - ▶ time, distance, speed
- For many problems, however, this isn't efficient.
- Organizing or consolidating the data in an intelligent way makes things much easier for programmers.

# Example 1: student database

Imagine that you've been given the job of preparing a database of SEE student ids. How should this be done?

There are various ways of doing this. We will see that each approach has its advantages and disadvantages.

*We will use this example to illustrate types of data structures.*

# Ex. 1 (cont): separate variables

The simplest solution is to have a separate variable for each student:

- `BillyChan = 1234`
- `CindyWong = 2345`
- `DerekMa = 3456`
- `EdithLee = 4567`

But it's tiresome to create a separate variable for each student. This approach doesn't lend itself to automation. We can't find the sid of an arbitrary student without knowing his/her name.

# Ex. 1 (cont): basic list

A better solution is to create a list of student ids

index	id
1	1234
2	2345
3	3456
4	4567

We can go through the entire list of students by changing the index

With this approach, it's easy to add additional students. But we've lost the names...



# Ex. 1 (cont): 2-D list

The names can be reintroduced by adding another column:

index	id	name
1	1234	Billy Chan
2	2345	Cindy Wong
3	3456	Derek Ma
4	4567	Edith Lee

We can still go through the entire list of students by changing the index. But we now have more information.

This is better. But working with the index can be inconvenient. For example, to find the id of a given student we need to find his or her index first.

# Ex. 1 (cont): lookup table

Sometimes it's convenient to get rid of the index:

name	id
Billy Chan	1234
Cindy Wong	2345
Derek Ma	3456
Edith Lee	4567

We can go through the list without an explicit index.

This is more direct. Once we know the name, the sid follows immediately. However, we can't jump to a specific student without knowing this information

# Data structures

- We've just seen that there are different ways of organizing data:
  - ▶ unstructured data
  - ▶ simple list
  - ▶ 2-d list
  - ▶ lookup table
- It's clear that the *data should be organized in different ways for different problems.*
- In the language of computer science, a **data structure** refers to the manner in which the data are organised.

these are data structures

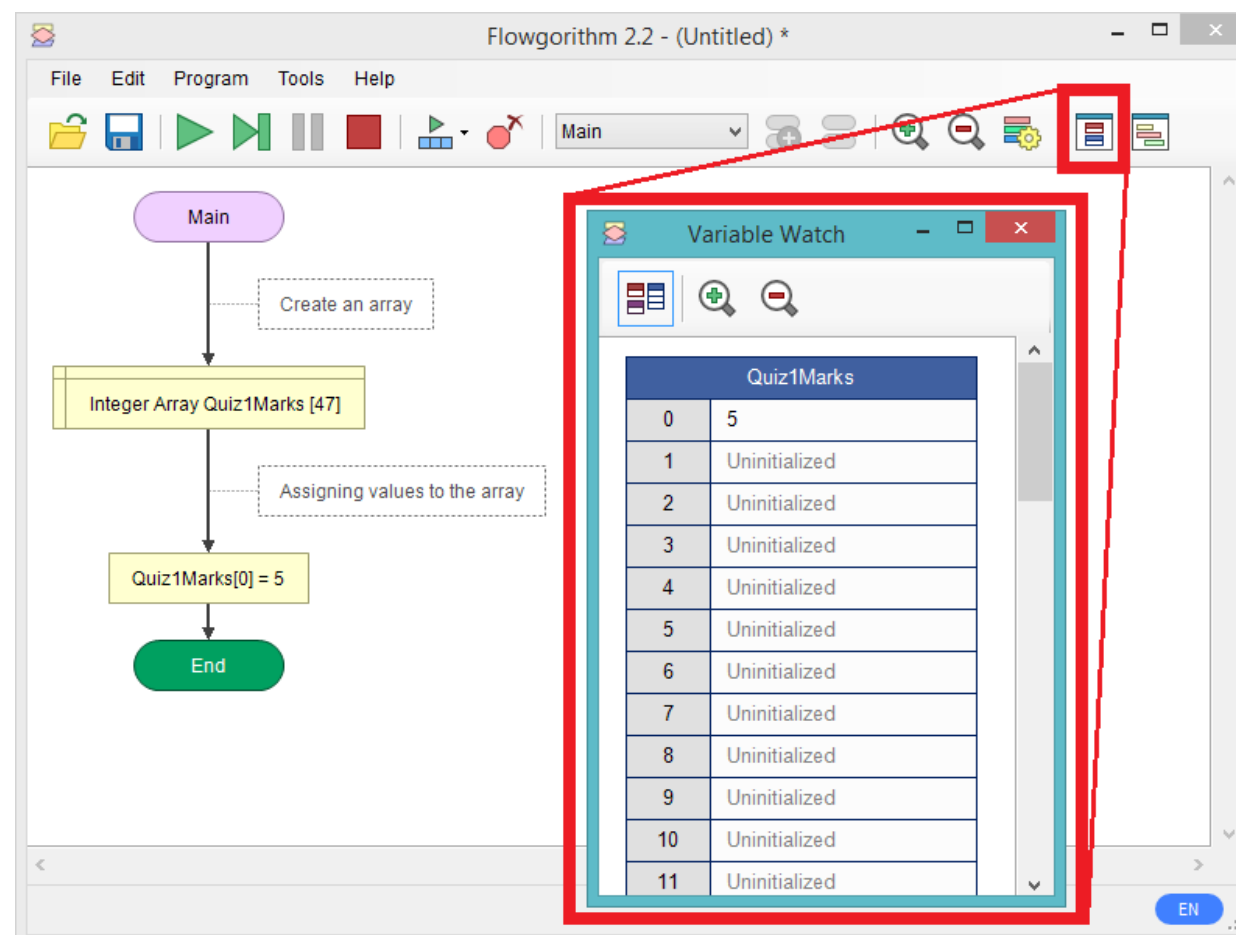
# Data structures in Python

- One of the nice things about Python is that it's easy to create data structures.
- For now, we'll only consider three data structures:
  1. List
  2. Tuple
  3. Dictionary
- Later in this course we'll consider more sophisticated data structures.

# Data structures in Flowgorithm

- Flowgorithm does not support as many data structures as Python.
- However, it does supports arrays. An **array** is a list composed of variables of identical data type.

Flowgorithm array



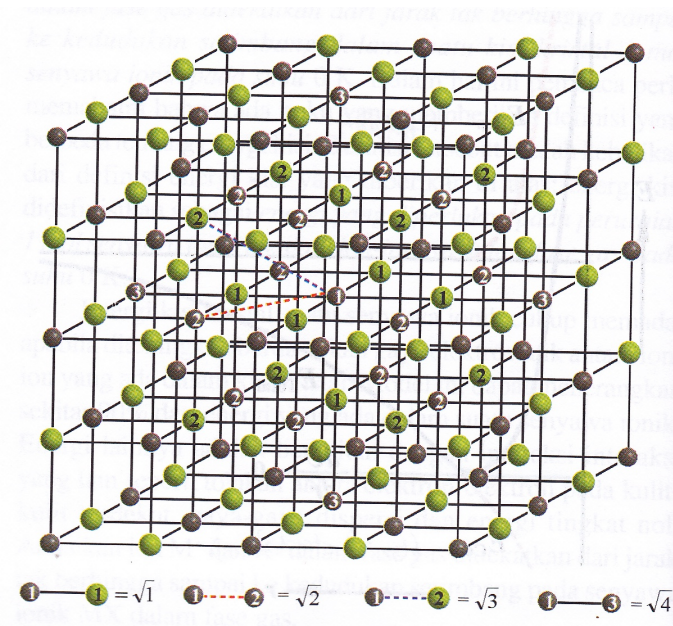
# Arrays in science and engineering

- Arrays are extremely useful in science and engineering!
- *Examples:*
  - Measurement of a variable (e.g. concentration) at different times.
  - Measurements of a field (e.g. pressure) at different locations.
- In the final part of this course, we will discuss the Python implementation of arrays.

## 2. Lists

# A. Lists

- The list is the **fundamental data structure** in Python.
- It is analogous to a **set** in mathematics.
- It represents a collection of data **indexed by an integer**.  
The index is associated with position and the values needn't be of the same type.
- In some cases it's natural to organize data using indexes.  
*Examples:* people waiting in a queue, points on a grid or a lattice.





# Examples of lists

- We have already encountered a few lists.
- Their defining characteristic is that they have no structure apart from their index.

index	id
1	1234
2	2345
3	3456
4	4567

If we know the starting index, we can store the id only.

# Lists in Python

- A Python list is just a collection of values enclosed by square brackets, [...]. Each member of the list is indexed by its position and is effectively a separate variable.
- The values do not need to be of the same type, i.e., we can mix integers, floats, strings, etc.
- One can create an empty list with []. This is analogous to the empty set in mathematics.

```
In [9]: list1=[]
```

```
In [10]: list1  
Out[10]: []
```

```
In [17]: list2=[1,2,3]
```

```
In [18]: list2  
Out[18]: [1, 2, 3]
```

```
In [19]: list3=['dog', 'cat', 'mouse']
```

```
In [20]: list3  
Out[20]: ['dog', 'cat', 'mouse']
```

```
In [22]: list4=[1,'dog',2,'cat',3,'mouse']
```

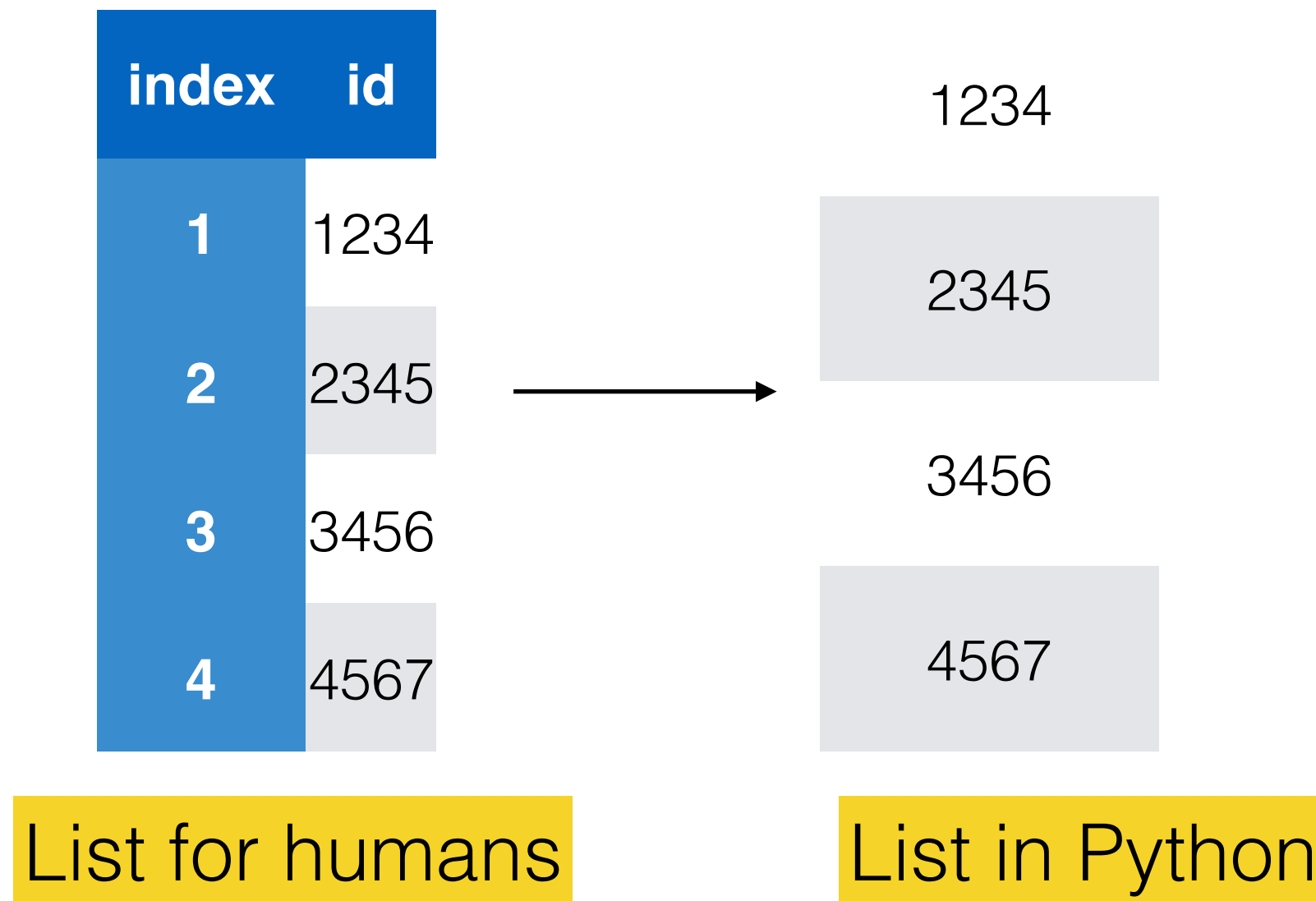
```
In [23]: list4  
Out[23]: [1, 'dog', 2, 'cat', 3, 'mouse']
```

```
In [11]: list5=[True, 'dog', 1.03, 'cat', 2, 'mouse']
```

```
In [12]: list5  
Out[12]: [True, 'dog', 1.03, 'cat', 2, 'mouse']
```

# Python lists vs. ordinary lists

Python lists have no column labels:



# Why are lists useful?

- The simple I-D list is easy to create and works with different kinds of data.
- Organization by index is straightforward and powerful.

*Examples:*

- ▶ To-do list
- ▶ Top students in a class
- ▶ Richest countries in the world

# Disadvantages

The main disadvantage is that it doesn't impose much structure. Sometimes a single index isn't very useful. It's a poor choice for certain datasets:

- ▶ Songs on phone or computer
- ▶ Student database
- ▶ Energy consumption of all the buildings in Hong Kong

We need other data structures to handle more complicated datasets.

# B. Tuples

- A tuple is basically the same as a list.
- It's nothing more than a collection of values enclosed by round brackets, ().
- Many of the operations on lists carry over to tuples.

```
In [13]: tuple1=()
```

```
In [14]: tuple1
```

```
Out[14]: ()
```

```
In [17]: tuple5=(True, 'dog', 1.03, 'cat', 2, 'mouse')
```

```
In [18]: tuple5
```

```
Out[18]: (True, 'dog', 1.03, 'cat', 2, 'mouse')
```

```
In [15]: tuple2=(1,2,3)
```

```
In [16]: tuple2
```

```
Out[16]: (1, 2, 3)
```

# Difference between tuples and lists

- Tuples differ from lists in one crucial aspect: they cannot be modified. The fancy word for this is **immutability**.

```
In [1]: tuple2=(1,2,3)
```

```
In [2]: tuple2
```

```
Out[2]: (1, 2, 3)
```

```
In [3]: tuple2[0]=4
```

```
-----  
TypeError                                Traceback (most recent call last)  
<ipython-input-3-bc5ca3b239fa> in <module>()  
----> 1 tuple2[0]=4
```

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

- Attempting to modify a tuple generates a **Type Error** because this is inconsistent with the data type.

# Why are tuples useful?

- Since lists and tuples are very similar, why would we want to use tuples?
- Tuples are useful in defining constants, i.e., values that can't change. This avoids problems created by accidentally changing the values in a list.
- *Example:* student id of students in a class, molecular weight of elements.



# C. List operations

There are many things that one can do with or to a list or tuple:

- i. Data access (*like retrieval*)
- ii. Data modification (*like assignment*)
- iii. Data queries (information about the list)

# I. Data access

- There are various ways of accessing the data in a list.
- The simplest is to refer to the list **as a whole**.

```
In [29]: list2=[1,2,3]
```

```
In [30]: list2  
Out[30]: [1, 2, 3]
```

Only works in shell

Works in programs

```
In [31]: print(list2)  
[1, 2, 3]
```

- This is not very useful! Imagine what happens when we have a big list but are only interested in a single number...

# Selective access

- The purpose of a list is to group together lots of information. We want to access this information **selectively**.
- This can be done **by** referencing the index, e.g.  
`mylist[index]`

```
In [7]: list2=[1,2,3]
```

```
In [12]: list2[0]  
Out[12]: 1
```

```
In [13]: list2[1]  
Out[13]: 2
```

```
In [14]: list2[2]  
Out[14]: 3
```

# Indexing rules

1. The index must be  $\leq N-1$  inclusive, where  $N$  is the length of the list.
2.  $N$  is automatically added to negative indexes, e.g.

`mylist[-1]=mylist[N-1]`

final element in list  
with  $N$  elements



3. Indexing starts at 0.

# Indexing convention

There are different ways of counting the elements of an array.

- The so-called C convention, which is used in Python, starts with **0**. It's favoured by computer scientists.
- The so-called Fortran convention starts with **1**. It's favoured by engineers and scientists. ***We will not use it in this course.***

# Access range

Sometimes it's useful to access a range of values. This can be done using `list[index1:index2]` which refers to elements from `index1` to `index2-1`.

```
In [7]: list2=[1,2,3]
```

```
In [21]: list2[0:1]  
Out[21]: [1]
```

```
In [22]: list2[0:2]  
Out[22]: [1, 2]
```

```
In [23]: list2[0:3]  
Out[23]: [1, 2, 3]
```

Note that `list[index1:index2]` has exactly `index2-index1` elements. Thus `list[0:index2]` has exactly `index2` elements.

# Connection with strings

You may recall that individual characters within a string can be accessed in the same way:

```
In [99]: var1='Homer'
```

```
In [100]: var1[0]
```

```
Out[100]: 'H'
```

```
In [101]: var1[1]
```

```
Out[101]: 'o'
```

```
In [102]: var1[0:5]
```

```
Out[102]: 'Homer'
```

In Python, *a string is equivalent to a list* (i.e., a list of characters). For our purposes, however, it's helpful to think of them as being distinct.

## II. Data modification

It's easy to change the value assigned to a variable:

```
In [94]: a=1
In [95]: print(a)
1
In [96]: a=2
In [97]: print(a)
2
```

We can do the same thing with members of a list:

```
In [98]: list2=[1,2,3]
In [99]: print(list2)
[1, 2, 3]
```

```
In [100]: list2[0]=4
In [101]: list2[1]=5
In [102]: list2[2]=6
```

```
In [104]: print(list2[0],list2[1],list2[2])
4 5 6
```

```
In [105]: print(list2)
[4, 5, 6]
```



# Constructing a list from scratch

- If you've been paying attention, you'll have noticed that we've constructed our lists from assignments., e.g.  
`list=[value1, value2,...]`. This is not very efficient if we have a big list!

# append

- The size of a list is not always known in advance.
- Examples:
  - students present for a class
  - student who want to go a field trip
  - etc.
- In some applications, it's more natural to construct a list by updating it as new information becomes available.
- We can do this using `append`.

# append syntax

- `list.append( )` adds an element to the end of a `list`.
- To construct a list incrementally:
  1. Use `[ ]` to create an empty list
  2. Each time a new piece of information becomes available, update the list using `append( )`

```
In [129]: mylist=[]  
In [130]: mylist.append(0)  
In [131]: mylist.append(1)  
In [132]: mylist.append(2)
```

```
In [134]: print(mylist)  
[0, 1, 2]
```

← add elements one at a time

# Why is append useful?

- It saves us from guessing how much memory we need.
- We can expand the size of a list as much as required based on the incoming data. This is referred to as **dynamic memory allocation**.
- In many applications, dynamic memory allocation is preferable to static allocation.

# Alternatives

- The `.append( )` syntax is easy to remember but it requires extra typing!
- After we learn about the `for` loop, we can use a shorter method.

# Other list functions

- `.append` is a list function. There are many other functions associated with lists (e.g. sorting).
- To see them, type tab after “`list.`” This is referred to as **tab completion**.
- Note that these functions are called by adding `.function` to the end of the list, where `.function` is the name of the function. Later in the the course, we’ll cover other functions that are called in exactly this way.
- *Note:* we’ll define functions later. They’re analogous to functions in mathematics.

# III. Data queries

- We also need to get information about a list.
- This is necessary because a list may be very long and we might not know what it contains.

*Example:*

- list containing SIDs for all of the students at CityU.

## a) len

The most important piece of information is the length of the list, which is returned by `len(list)`. This is an integer corresponding to the number of elements in the list.

```
In [147]: mylist=[0,1,2]
```

```
In [148]: len(mylist)
```

```
Out[148]: 3
```

```
In [149]: mylist2=[]
```

```
In [150]: len(mylist2)
```

```
Out[150]: 0
```



## b) in

- We also need to determine whether an item belongs to a list.
- This can be done using `in`:

```
In [170]: list3  
Out[170]: [1, 2, 3]
```

```
In [171]: 1 in list3  
Out[171]: True
```

```
In [180]: 4 in list3  
Out[180]: False
```

- `in` returns a Boolean variable.
- `in` can save a lot of time! Note that it's analogous to list membership in mathematics, e.g.,  $x \in \mathbb{R}$  or  $x \in \{\dots\}$

# 3. Dictionaries

# Dictionaries

- Lists are convenient, but they have one disadvantage: the values are associated with an **index**.
- In some cases, it's more *natural to forget about the index*.
- This is how real **dictionaries** work. We don't care about a word's position within the dictionary: what's important is that there is a definition associated with it.

dictionary →

A **dictionary** is a collection of **words** in one or more specific languages, often arranged **alphabetically** (or by **radical and stroke** for **ideographic** languages), which may include information on **definitions**, usage, **etymologies**, **phonetics**, **pronunciations**, translation, etc.<sup>[1]</sup> or a book of words in one language with their equivalents in another, sometimes known as a **lexicon**.<sup>[1]</sup> It is a **lexicographical** product which shows inter-relationships among the data.<sup>[2]</sup>

# How do dictionaries work?

- In **real dictionaries**, a *keyword* is mapped to a *definition*.
- In **Python**, a **key** is mapped to a **value**.
  - ▶ For certain kinds of data it's more convenient to associate a value with a label or **key** instead of a number.
- Comparison with a list:
  - *index*: 0 → 3.1415, 1 → 2.714, 2 → 9.81 `list=[3.1415,2.714,9.81]`
  - *key*: pi → 3.1415, e → 2.714, g → 9.81 

We will turn this into a dictionary!

# Connection with lookup table

We have already seen a **dictionary data structure**. But previously we called it a lookup table:

name	id
Billy Chan	1234
Cindy Wong	2345
Derek Ma	3456
Edith Lee	4567

constant	value
pi	3.1415
e	2.714
g	9.81

# Difference with respect to lookup table

Python doesn't require column labels:

name	id
Billy Chan	1234
Cindy Wong	2345
Derek Ma	3456
Edith Lee	4567

→

Billy Chan	1234
Cindy Wong	2345
Derek Ma	3456
Edith Lee	4567

key value

# Using Python dictionaries

- A dictionary can be created analogously to a list, but using **braces** rather than square brackets:

```
In [98]: telephone = {'patrick': 1234, 'denis': 2345, 'alicia': 3456}
```

```
In [99]: telephone
```

```
Out[99]: {'alicia': 3456, 'denis': 2345, 'patrick': 1234}
```

- It's accessed by referring to the **key** rather than the index:

```
In [100]: telephone['alicia']
```

```
Out[100]: 3456
```

```
In [101]: telephone['denis']
```

```
Out[101]: 2345
```

```
In [103]: telephone['patrick']
```

```
Out[103]: 1234
```

key



- Each *key-value* pair is represented in Python as

```
element={key:value}
```

# Key Error

If we attempting to reference a key that hasn't been defined, this will generate an error message.

```
In [161]: characters={'Donald':'Duck', 'Minnie':'Mouse', 'Hello':'Kitty','Oscar': 'The Grouch'}
```

```
In [165]: print(characters['Mickey'])  
Traceback (most recent call last):
```

```
File "<ipython-input-165-a59b426ec5e6>", line 1, in <module>  
    print(characters['Mickey'])
```

```
KeyError: 'Mickey'
```

missing key

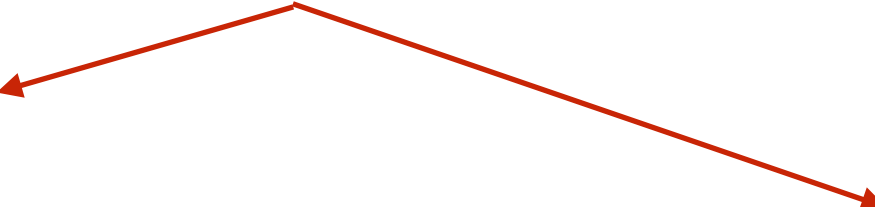




# Modifying a dictionary

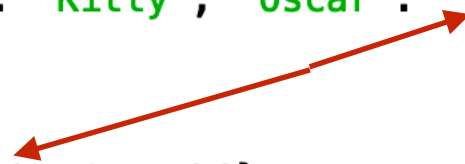
- Each element of a Python dictionary can be modified in exactly the same way as any Python variable:

```
In [161]: characters={'Donald':'Duck', 'Minnie':'Mouse', 'Hello':'Kitty', 'Oscar': 'The Grouch'}  
  
In [166]: characters['Oscar']='the Grouch'  
  
In [169]: characters  
Out[169]: {'Donald': 'Duck', 'Minnie': 'Mouse', 'Hello': 'Kitty', 'Oscar': 'the Grouch'}  
  
In [170]: print( characters['Oscar'] )  
the Grouch
```



- Alternatively we can recreate the dictionary from scratch:

```
In [171]: characters={'Donald': 'Duck', 'Minnie': 'Mouse', 'Hello': 'Kitty', 'Oscar': 'the Grouch'}  
  
In [172]: print(characters)  
{'Donald': 'Duck', 'Minnie': 'Mouse', 'Hello': 'Kitty', 'Oscar': 'the Grouch'}
```



# update

- A Python dictionary can be expanded or updated by adding new items to it.

```
In [173]: characters.update( {'Elmer': 'Fudd'} )
```

```
In [174]: print(characters)
{'Donald': 'Duck', 'Minnie': 'Mouse', 'Hello': 'Kitty', 'Oscar': 'the Grouch', 'Elmer': 'Fudd'}
```

- The update is equivalent to `dict.update(element)` or `dict.update( {key:value} )` .

# Summary

1. Data can be consolidated using lists and dictionaries.
2. This list is the fundamental data structure in Python. It's useful for data without much structure.
3. Lists can be extended using `append`.
4. A dictionary associates values with a key rather than an index.