Data Structures for Programming with Python

CS101 lectures : part 3

Professor Lilian Besson

Mahindra École Centrale (School of Computer Science)

March 16th, 2015

Mahindra
École Centrale

Please contact me by email if needed: CS101@crans.org
Slides and examples will be uploaded on **Moodle**.

Overview of the content of these lectures

1. Presentation
2. A quick and interactive sum-up of the first two months
   - A game in Python ? 3 minutes of fun!
3. Data structures for programming cleverly (3 lectures)
   - Lists in Python (1/3)
   - Sets in Python (2/3)
   - Dictionaries in Python (3/3)
4. Sum-um about data structures : lists, sets, and dicts

Stay focus, listen and take notes

If you talk or disturb my lectures, I will kick you out *immediately*.
OK ?

Overview of the content of these lectures

1. Presentation
2. A quick and interactive sum-up of the first two months
   - A game in Python ? 3 minutes of fun!
3. Data structures for programming cleverly (3 lectures)
   - Lists in Python (1/3)
   - Sets in Python (2/3)
   - Dictionaries in Python (3/3)
4. Sum-um about data structures : lists, sets, and dicts

Stay focus, listen and take notes

If you talk or disturb my lectures, I will kick you out *immediately*. OK ?

A multi-player "Bomberman" game written in Python

During 3 minutes, try to see in this demo **where** and **how** each of the studied concepts are used:

Concepts already studied in CS101 lectures and labs

- **Variables**,
- **Arithmetical operations**,
- **Printing** and **interaction** with the user,
- **Conditional tests** (`if-elif-else`),
- **Conditional loops** (`while`),
- **Lists** and `for` **loops** over a list,
- ...
- (and many more things were used)

Definition and concept of a list in Python

Concept of a list and definition *(not new)*

"A list is a (finite) ***ordered sequence*** *of* ***any*** *kind of Python* ***values****."*

Goal ?

$\longrightarrow$ efficiently and easily **store** any kind of data, that can be **accessed** and **modified** later.

Practice by yourself (online)?

You should try to **practice a little bit by yourself**, *additionally* to the labs. That page IntroToPython.org/lists_tuples.html contains great exercises!

Definition and concept of a list in Python

Concept of a list and definition *(not new)*

"A list is a (finite) ***ordered sequence*** *of* ***any*** *kind of Python* ***values****."*

Goal ?

$\longrightarrow$ efficiently and easily **store** any kind of data, that can be **accessed** and **modified** later.

Practice by yourself (online)?

You should try to **practice a little bit by yourself**, *additionally* to the labs. That page IntroToPython.org/lists_tuples.html contains great exercises!

Defining, accessing, modifying a list

Defining a list and reading its elements *(not new)*

- it is easy to **define a list**:
  - empty `l = []`,
  - or not `l = [3, 4, 5]`, `team = ['Rahul', 'Nikitha']` etc,
- and to **read** its elements: `l[0]`, ..., `l[n-1]`
  (if `n = len(l)` is the **length** of the list, ie. its size)).

Two important **warnings** *(new!)*

- indexing **starts from** 0 **to** $n-1$ and not from 1 !
- indexing errors can (and will) happen: for $k > n-1$,
  `l[k]` raises a `IndexError: list index out of range` .

Modifying a list in place *(not new)*

- **modifying one element**: `l[0] = 6` makes `l` becoming `[6, 4, 5]`,
- modifying a **slice** (a sub-list) will be seen after.

Defining, accessing, modifying a list

Defining a list and reading its elements *(not new)*

- it is easy to **define a list**:
  - empty `l = []`,
  - or not `l = [3, 4, 5]`, `team = ['Rahul', 'Nikitha']` etc,
- and to **read** its elements: `l[0]`, ..., `l[n-1]`
  (if `n = len(l)` is the **length** of the list, ie. its size)).

Two important **warnings** *(new!)*

- indexing **starts from** 0 **to** $n-1$ and not from 1 !
- indexing errors can (and will) happen: for $k > n-1$,
  `l[k]` raises a `IndexError: list index out of range` .

Modifying a list in place *(not new)*

- **modifying one element**: `l[0] = 6` makes `l` becoming `[6, 4, 5]`,
- modifying a **slice** (a sub-list) will be seen after.

One classic example of lists: arithmetical progressions

`range` creates (finite) arithmetical progressions *(not new)*

The `range` function can be used, in three different ways:

- `range(n)` $= [0, 1, .., n-1]$,
- `range(a, b)` $= [a, a+1, .., b-1]$,
- `range(a, b, k)` $= [a, a+k, a+2k, .., a+i \times k]$ (last i with $a + ik < b$).

Default values are $a = 0$ and $k = 1$, and $k \neq 0$ is required.

Useful for loops! *(not new)*

For example, to print the square of the first 30 odd integers:

```
for i in range(1, 31, 2):
    # 1 <= i < 1 + 2*(31/2) - 1
    print i, "**2 is", i**2
```

Remark: if we just use `range` in a **for** loop, the `xrange` function is better (more time and memory efficient).

One classic example of lists: arithmetical progressions

`range` creates (finite) arithmetical progressions *(not new)*

The `range` function can be used, in three different ways:

- `range(n)` $= [0, 1, .., n-1]$,
- `range(a, b)` $= [a, a+1, .., b-1]$,
- `range(a, b, k)` $= [a, a+k, a+2k, .., a+i \times k]$ (last i with $a + ik < b$).

Default values are $a = 0$ and $k = 1$, and $k \neq 0$ is required.

Useful for loops! *(not new)*

For example, to print the square of the first 30 odd integers:

```
for i in range(1, 31, 2):
    # 1 <= i < 1 + 2*(31/2) - 1
    print i, "**2 is", i**2
```

Remark: if we just use `range` in a **for** loop, the `xrange` function is better (more time and memory efficient).

Looping over a list

To loop over a list, there is 3 approaches *(new!)*

- `for i in range(len(l)):` *(not new)*
  then the values of `l` can be obtained with `l[i]` (one by one),
- `for x in l:` *(not new)*
  then the values of `l` are just `x` (one by one),
- `for i, x in enumerate(l):` *(new!)*
  then the values of `l` are just `x` (one by one),
  but can be modified with `l[i] = newvalue`
- Warning: modifying this `x` will **not** change the list!

Example of a simple `for` loop

```
for name in ['Awk Girl', 'Batman', 'Wonder Woman']:
    print name, "is a member of the JLA."
```

Looping over a list

To loop over a list, there is 3 approaches *(new!)*

- `for i in range(len(l)):` *(not new)*
  then the values of `l` can be obtained with `l[i]` (one by one),
- `for x in l:` *(not new)*
  then the values of `l` are just `x` (one by one),
- `for i, x in enumerate(l):` *(new!)*
  then the values of `l` are just `x` (one by one),
  but can be modified with `l[i] = newvalue`
- Warning: modifying this `x` will **not** change the list!

Example of a simple `for` loop

```
for name in ['Awk Girl', 'Batman', 'Wonder Woman']:
    print name, "is a member of the JLA."
```

Negative indexing and slicing for a list

Negative indexing of a list `l`? *(new!)*

We can read its elements **from the end** with negative indexes.

- Instead of writing `l[n-1]`, write `l[-1]`: it is simpler!
- Similarly `l[-2]` is like `l[n-2]` etc.

Warning: indexing errors can still happen, `l[-k]` raises a `IndexError: list index out of range` when $k > n$.

Slicing for a list? `l` *(new!)*

Slicing a list is useful to **select a sub-list** of the list: `l[first:bound:step]`. By default, `first` is 0 (inclusive), `bound` is n (exclusive), `step` is 1.

- reading a slice: `l[0:3]`, `l[2:]` or `l[:3]` or `l[0::2]` for examples.
- modifying a slice: `l[:5] = [0]*5` for example puts a 0 in each of the 5 values `l[i]` for $0 \leqslant i < 5$,
- Warning: modifying a slice with a list of different size **might** raise an error like `ValueError: attempt to assign sequence of size 5 to extended slice of size 4` (it is a tricky point, be cautious).

Negative indexing and slicing for a list

Negative indexing of a list `l`? *(new!)*

We can read its elements **from the end** with negative indexes.

- Instead of writing `l[n-1]`, write `l[-1]`: it is simpler!
- Similarly `l[-2]` is like `l[n-2]` etc.

Warning: indexing errors can still happen, `l[-k]` raises a `IndexError: list index out of range` when $k > n$.

Slicing for a list? `l` *(new!)*

Slicing a list is useful to **select a sub-list** of the list: `l[first:bound:step]`. By default, `first` is 0 (inclusive), `bound` is n (exclusive), `step` is 1.

- reading a slice: `l[0:3]`, `l[2:]` or `l[:3]` or `l[0::2]` for examples.
- modifying a slice: `l[:5] = [0]*5` for example puts a 0 in each of the 5 values `l[i]` for $0 \leqslant i < 5$,
- Warning: modifying a slice with a list of different size **might** raise an error like `ValueError: attempt to assign sequence of size 5 to extended slice of size 4` (it is a tricky point, be cautious).

Some functions for lists

We give here some **functions for lists**, already seen and used in labs.

5 useful functions *(not new)*

- `len` gives the *length* of the list (and `len([]) = 0`),
- `min` and `max` returns the *minimum* and the *maximum* of the list.
  Might raise `ValueError: min() arg is an empty sequence`.
- `sum` computes the *sum* of the values in the list.
  Warning: there is **no `prod`** function to compute the product!
- `sorted` sorts the list (if possible, in $O(n \log(n))$ in the worst case), and returns **a new copy** of the list, sorted in the increasing order.

And more functions are available!

- `all` (resp. `any`) computes the **boolean** $\forall x \in mylist$ (resp. $\exists x \in mylist$),
- `filter` and `map` are not really used in practice,
- and `reduce` is ... more complicated.

Some functions for lists

We give here some **functions for lists**, already seen and used in labs.

5 useful functions *(not new)*

- `len` gives the *length* of the list (and `len([]) = 0`),
- `min` and `max` returns the *minimum* and the *maximum* of the list.
  Might raise `ValueError: min() arg is an empty sequence`.
- `sum` computes the *sum* of the values in the list.
  Warning: there is **no `prod`** function to compute the product!
- `sorted` sorts the list (if possible, in $O(n \log(n))$ in the worst case), and returns **a new copy** of the list, sorted in the increasing order.

And more functions are available!

- `all` (resp. `any`) computes the **boolean** $\forall x \in mylist$ (resp. $\exists x \in mylist$),
- `filter` and `map` are not really used in practice,
- and `reduce` is ... more complicated.

Some methods for lists

2 convenient notations: *(new!)*

- `l1 + l2` is the concatenation of the two lists `l1` and `l2`,
- `l * k` is like `l + l + ... + l`, k times. Example: `l = [0] * 100`.

Some **methods** for lists can be useful.

7 simple methods: *(new!)*

- `l.sort()` sorts the list `l` **in place** (ie. modifies the list),
- `l.append(newvalue)` adds the value `newvalue` at the end
- `l.pop()` removes and returns the last item,
- `l.index(x)` returns the first index of value `x`.
  Will raise `ValueError` if the value `x` is not present in `l`,
- `l.count(y)` counts how many times the value `y` is present,
- `l.extend(otherlist)` is like `l = l + otherlist`,
- `l.insert(index, z)` will insert the new value `z` at position `index`.

Some methods for lists

2 convenient notations: *(new!)*

- `l1 + l2` is the concatenation of the two lists `l1` and `l2`,
- `l * k` is like `l + l + ... + l`, k times. Example: `l = [0] * 100`.

Some **methods** for lists can be useful.

7 simple methods: *(new!)*

- `l.sort()` sorts the list `l` **in place** (ie. modifies the list),
- `l.append(newvalue)` adds the value `newvalue` at the end
- `l.pop()` removes and returns the last item,
- `l.index(x)` returns the first index of value `x`.
  Will raise ValueError if the value `x` is not present in `l`,
- `l.count(y)` counts how many times the value `y` is present,
- `l.extend(otherlist)` is like `l = l + otherlist`,
- `l.insert(index, z)` will insert the new value `z` at position `index`.

Sum-up about **lists**, and what for tomorrow?

About lists, we saw: *(new!)*

- concept of a list in Python, and how to define it, modify it or its elements and read them,
- some new concepts, like **negative indexing** or **slicing**,
- looping, 3 approaches, and `enumerate(l)` is new,
- functions for lists (`len`, `max`/`min`, `sum`, `sorted` ...),
- methods for lists (`sort`, `append`, `pop`, `index`, `count`, `extend` etc ...).

What is next ?

Tomorrow: matrices as **list of line vectors**, some more list comprehensions, and one nice example will be seen (with a *list* of your *grades*).
We will then introduce **sets** in Python, like `s = { -1, 1 }`.

Sum-up about **lists**, and what for tomorrow?

About lists, we saw: *(new!)*

- concept of a list in Python, and how to define it, modify it or its elements and read them,
- some new concepts, like **negative indexing** or **slicing**,
- looping, 3 approaches, and `enumerate(l)` is new,
- functions for lists (`len`, `max`/`min`, `sum`, `sorted` ...),
- methods for lists (`sort`, `append`, `pop`, `index`, `count`, `extend` etc ...).

What is next ?

Tomorrow: matrices as **list of line vectors**, some more list comprehensions, and one nice example will be seen (with a *list* of your *grades*).
We will then introduce **sets** in Python, like `s = { -1, 1 }`.

Outline for today (March 10th) Lecture 2/3

1. Presentation
2. A quick and interactive sum-up of the first two months
3. Data structures for programming cleverly (3 lectures)
   - Lists in Python (1/3)
   - Sets in Python (2/3)
   - Dictionaries in Python (3/3)
4. Sum-um about data structures : lists, sets, and dicts

Introduction to **list comprehensions**

Python offers a nice and efficient syntax for **easily defining a list of values** obtained with an expression of one index.

What is a **list comprehension**? *(new!)*

The syntax is like this: `[` "expression with i" `for i in somelist ]`

- List of the first 100 triangular numbers?
  ↪ `[ k*(k+1)/2 for k in range(100) ]`,

- Quickly compute a partial sum of a series? $S_{10000} = \sum_{n=1}^{10000} 1/n^2$
  ↪ `S = sum([ 1.0/(n**2) for n in range(1,10000+1) ])`,

- Sum of numbers below 10000 that are multiples of 11 **or** multiples of 7?
  ↪ `sum([ k for k in range(1,10000) if k%7==0 or k%11==0 ])`

- and many more examples are possible . . .

Introduction to **list comprehensions**

Python offers a nice and efficient syntax for **easily defining a list of values** obtained with an expression of one index.

What is a **list comprehension**? *(new!)*

The syntax is like this: `[` "expression with i" `for i in somelist ]`

- List of the first 100 triangular numbers?
  ↪ `[ k*(k+1)/2 for k in range(100) ]`,

- Quickly compute a partial sum of a series? $S_{10000} = \sum_{n=1}^{10000} 1/n^2$
  ↪ `S = sum([ 1.0/(n**2) for n in range(1,10000+1) ])`,

- Sum of numbers below 10000 that are multiples of 11 **or** multiples of 7?
  ↪ `sum([ k for k in range(1,10000) if k%7==0 or k%11==0 ])`

- and many more examples are possible . . .

Introduction to **list comprehensions**

Python offers a nice and efficient syntax for **easily defining a list of values** obtained with an expression of one index.

What is a **list comprehension**? *(new!)*

The syntax is like this: `[` "expression with i" `for i in somelist ]`

- List of the first 100 triangular numbers?
  ↪ `[ k*(k+1)/2 for k in range(100) ]`,

- Quickly compute a partial sum of a series? $S_{10000} = \sum_{n=1}^{10000} 1/n^2$
  ↪ `S = sum([ 1.0/(n**2) for n in range(1,10000+1) ])`,

- Sum of numbers below 10000 that are multiples of 11 **or** multiples of 7?
  ↪ `sum([ k for k in range(1,10000) if k%7==0 or k%11==0 ])`

- and many more examples are possible ...

Matrices as list of lists

Matrix as list of line vectors *(new!)*

$M = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$ has three lines, $x_1 = [1\ 2\ 3], x_2 = [4\ 5\ 6]$, $x_3 = [7\ 8\ 9]$.

So in Python, this matrix can be written as `M = [x_1, x_2, x_3]`, with:
`M[0]=x_1=[1, 2, 3]`, `M[1]=x_2=[4, 5, 6]`, `M[2]=x_3=[7, 8, 9]`.

`M = [ [1, 2, 3], [4, 5, 6], [7, 8, 9] ]` is a list of 3 lists (of sizes 3).

Examples of list comprehension for matrices *(new!)*

- trace(M) is `sum([ M[i][i] for i in range(len(M)) ])`,
- $A + B$ is `[ [ A[i][j] + B[i][j] for i in range(len(A)) ] for j in range(len(A)) ]` (if A and B are square),
- ... and you will figure out $A \times B$ by yourself (in this week lab).

Matrices as list of lists

Matrix as list of line vectors *(new!)*

$M = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$ has three lines, $x_1 = [1\ 2\ 3], x_2 = [4\ 5\ 6]$, $x_3 = [7\ 8\ 9]$.

So in Python, this matrix can be written as `M = [x_1, x_2, x_3]`, with:
`M[0]=x_1=[1, 2, 3]`, `M[1]=x_2=[4, 5, 6]`, `M[2]=x_3=[7, 8, 9]`.

`M = [ [1, 2, 3], [4, 5, 6], [7, 8, 9] ]` is a list of 3 lists (of sizes 3).

Examples of list comprehension for matrices *(new!)*

- trace(M) is `sum([ M[i][i] for i in range(len(M)) ])`,
- $A + B$ is `[ [ A[i][j] + B[i][j] for i in range(len(A)) ] for j in range(len(A)) ]` (if A and B are square),
- ... and you will figure out $A \times B$ by yourself (in this week lab).

Matrices as list of lists

Matrix as list of line vectors *(new!)*

$M = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$ has three lines, $x_1 = [1\ 2\ 3], x_2 = [4\ 5\ 6]$, $x_3 = [7\ 8\ 9]$.

So in Python, this matrix can be written as `M = [x_1, x_2, x_3]`, with:
`M[0]=x_1=[1, 2, 3]`, `M[1]=x_2=[4, 5, 6]`, `M[2]=x_3=[7, 8, 9]`.

`M = [ [1, 2, 3], [4, 5, 6], [7, 8, 9] ]` is a list of 3 lists (of sizes 3).

Examples of list comprehension for matrices *(new!)*

- trace(M) is `sum([ M[i][i] for i in range(len(M)) ])`,
- $A + B$ is `[ [ A[i][j] + B[i][j] for i in range(len(A)) ] for j in range(len(A)) ]` (if A and B are square),
- ... and you will figure out $A \times B$ by yourself (in this week lab).

Other data structures similar to list : tuples

About tuples

Tuples are exactly like lists,
but **cannot be modified after being created**:

- A **tuple** is an **unmutable list**.
- A tuple is written `s = ()` for the *empty* tuple, or `t = (x, y)` For example, `v = (1, 0, 1)` is like a vector of $\mathbb{R}^3$.
- **Type conversion** between tuples and lists can be done: with `t = tuple(mylist)` and `l = list(mytuple)` ... !

Other data structures similar to list : strings

About strings

A **string** is *almost* like a **list of characters**:
`name = 'batman'` is like `['b', 'a', 't', 'm', 'a', 'n']`:

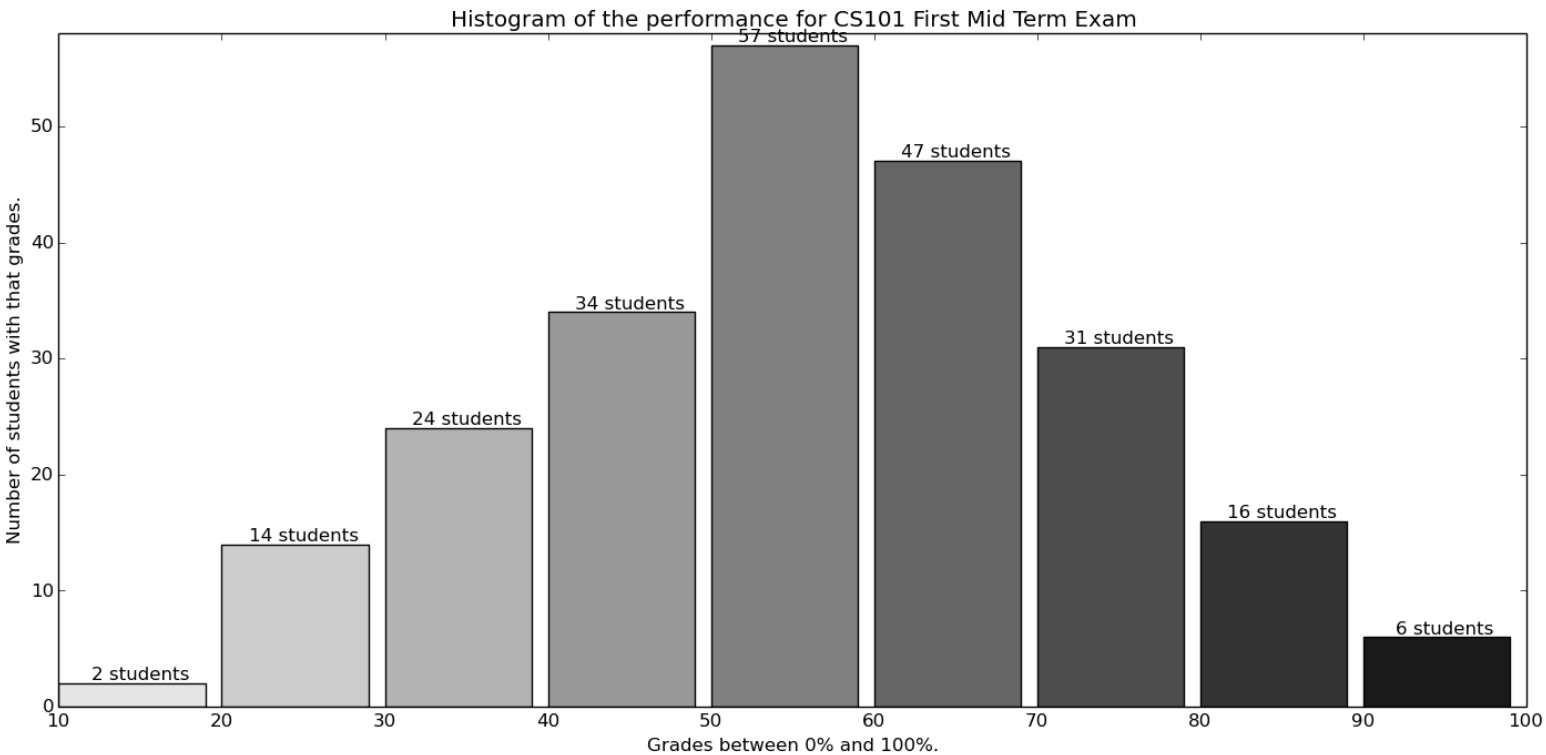
- Accessing and slicing is done the same way for **strings**:
  `name[0]` is `'b'`, `name[3:]` is `'man'` etc,
- Looping over a string will loop **letter by letter** (`'b'`,`'a'` etc).
- But warning: a string is **unmutable**!
  `name[0] = 'C'` fails, `name = 'C' + name[1:]` is good

One example of use of a list (cf. Spyder demo)

Using a list to store values, e.g. grades of the first Mid Term Exam

The demo plots an histogram for your grades, written as a Python list:

```
# That list has 230 values between 0 and 100
grades = [ 36, 73.5, ..., 34, 56, 68, 61, 29 ]
```



One first example of the use of sets in Python

First CS Lab Examination : problem 1 (1/2)

For the sum of multiples of 7 **or** 11 below 10000, one line is enough:

```
sum( set(xrange(7,10000,7)) | set(xrange(11,10000,11)) ).
```

First CS Lab Examination : problem 1 (2/2)

And for the product of multiples of 3 **or** 5 below 30, one line also:

```
prod( set(xrange(3,30,3)) | set(xrange(5,30,5)) ).
```

It requires to have a function `prod`, easily defined like this:

```
def prod(mylist):
    p = 1
    for value in mylist:
        p *= value
    return p
```

One first example of the use of sets in Python

First CS Lab Examination : problem 1 (1/2)

For the sum of multiples of 7 **or** 11 below 10000, one line is enough:

```
sum( set(xrange(7,10000,7)) | set(xrange(11,10000,11)) ).
```

First CS Lab Examination : problem 1 (2/2)

And for the product of multiples of 3 **or** 5 below 30, one line also:

```
prod( set(xrange(3,30,3)) | set(xrange(5,30,5)) ).
```

It requires to have a function `prod`, easily defined like this:

```
def prod(mylist):
    p = 1
    for value in mylist:
        p *= value
    return p
```

Definition and concept of a set in Python

Concept of a set and definition *(not new)*

"A set is a (finite) ***unordered and mutable collection*** *of* ***unique*** *Python* ***values****."*

Goal ?

Efficiently and easily **store** and represent (finite) mathematical sets.
Examples: `s = {}` the empty set $\varnothing$, or `u = {-1, 1}`.

Limitation ?

Python has to be able to say if two elements are equal or not (to keep only unique values).
Therefore, elements of a set need to be **hashable**, ie. the `hash` function can be used (this will be seen later).

Definition and concept of a set in Python

Concept of a set and definition *(not new)*

"A set is a (finite) ***unordered and mutable collection*** *of* ***unique*** *Python* ***values****."*

Goal ?

Efficiently and easily **store** and represent (finite) mathematical sets.
Examples: `s = {}` the empty set $\varnothing$, or `u = {-1, 1}`.

Limitation ?

Python has to be able to say if two elements are equal or not (to keep only unique values).
Therefore, elements of a set need to be **hashable**, ie. the `hash` function can be used (this will be seen later).

Definition and concept of a set in Python

Concept of a set and definition *(not new)*

"A set is a (finite) ***unordered and mutable collection*** *of* ***unique*** *Python* ***values****."*

Goal ?

Efficiently and easily **store** and represent (finite) mathematical sets.
Examples: `s = {}` the empty set $\varnothing$, or `u = {-1, 1}`.

Limitation ?

Python has to be able to say if two elements are equal or not (to keep only unique values).
Therefore, elements of a set need to be **hashable**, ie. the `hash` function can be used (this will be seen later).

Basic notions about sets in Python

Cardinality for a set *(new!)*

- `len(s)`: **cardinality** of set `s`.

Testing **membership** *(new!)*

- `x in s`: test `x` for **membership** in `s` ($x \in s$?).
- `x not in s`: test `x` for **non-membership** in `s` ($x \notin s$?).

Simple examples

- `len({-1, 1})` is ?
- `0 in {1, 2, 3}` is ?
- `'ok' not in {'ok', 'right', 'clear'}` is ?

Basic notions about sets in Python

Cardinality for a set *(new!)*

- `len(s)`: **cardinality** of set `s`.

Testing **membership** *(new!)*

- `x in s`: test `x` for **membership** in `s` ($x \in s$?).
- `x not in s`: test `x` for **non-membership** in `s` ($x \notin s$?).

Simple examples

- `len({-1, 1})` is ?
- `0 in {1, 2, 3}` is ?
- `'ok' not in {'ok', 'right', 'clear'}` is ?

Comparing sets: containing, being contained etc

Easy syntax: we use Python comparison operators: `<`, `<=`, `>`, `>=`, `==`, `!=`.

Is `other` a sub-set or super-set or `s` *(new!)*

- `s <= other` (or `s.issubset(other)`): ($s \subseteq$ other ?).
  every element in `s` is in `other` ? (**subset**).
- `s < other`: ($s \subset$ other ?).
  `s` is a *proper* subset of `other` ?, ie. `s <= other` *and* `s != other`.
- `s >= other` (or `s.issuperset(other)`): ($s \supseteq$ other ?).
  every element in `other` is in `s` ? (**superset**).
- `s > other`: ($s \subset$ other ?).
  `s` is a *proper* superset of `other` ?, ie. `s >= other` *and* `s != other`.

Check if two set are **disjoint** *(new!)*

`s.isdisjoint(other)` is `True` if they have no elements in common.
Remark: `s` and `other` are **disjoint** $\Leftrightarrow$ their intersection `s & other` is the empty set `{}`.

Comparing sets: containing, being contained etc

Easy syntax: we use Python comparison operators: `<`, `<=`, `>`, `>=`, `==`, `!=`.

Is `other` a sub-set or super-set or `s` *(new!)*

- `s <= other` (or `s.issubset(other)`): ($s \subseteq$ other ?).
  every element in `s` is in `other` ? (**subset**).
- `s < other`: ($s \subset$ other ?).
  `s` is a *proper* subset of `other` ?, ie. `s <= other` *and* `s != other`.
- `s >= other` (or `s.issuperset(other)`): ($s \supseteq$ other ?).
  every element in `other` is in `s` ? (**superset**).
- `s > other`: ($s \subset$ other ?).
  `s` is a *proper* superset of `other` ?, ie. `s >= other` *and* `s != other`.

Check if two set are **disjoint** *(new!)*

`s.isdisjoint(other)` is `True` if they have no elements in common.
Remark: `s` and `other` are **disjoint** $\Leftrightarrow$ their intersection `s & other` is the empty set `{}`.

Comparing sets: containing, being contained etc

Easy syntax: we use Python comparison operators: `<`, `<=`, `>`, `>=`, `==`, `!=`.

Is `other` a sub-set or super-set or `s` *(new!)*

- `s <= other` (or `s.issubset(other)`): ($s \subseteq$ other ?).
  every element in `s` is in `other` ? (**subset**).
- `s < other`: ($s \subset$ other ?).
  `s` is a *proper* subset of `other` ?, ie. `s <= other` *and* `s != other`.
- `s >= other` (or `s.issuperset(other)`): ($s \supseteq$ other ?).
  every element in `other` is in `s` ? (**superset**).
- `s > other`: ($s \subset$ other ?).
  `s` is a *proper* superset of `other` ?, ie. `s >= other` *and* `s != other`.

Check if two set are **disjoint** *(new!)*

`s.isdisjoint(other)` is `True` if they have no elements in common.
Remark: `s` and `other` are **disjoint** $\Leftrightarrow$ their intersection `s & other` is the empty set `{}`.

Math. operations on sets: union, intersection etc

A complicated syntax ?

Syntax is natural here again: `|`, `&`, `-`, $\wedge$.

4 operations coming from maths ($\cup, \cap, \setminus, \Delta$) *(new!)*

Create new set with elements. . . :

- `s.union(other, ...)` or `s | other | ...`
  from the set `s` and all `others`. $(s \cup s_2 \cup \dots)$
- `s.intersection(other, ...)` or `s & other & ...`
  common to the set `s` and all `others`. $(s \cap s_2 \cap \dots)$
- `s.difference(other, ...)` or `s - other - ...`
  in the set `s` that are not in the `others`. $((s \setminus s_2) \setminus s_3 \dots)$
- `s.symmetric_difference(other)` or `s ^ other` *(new!)*
  in either the set `s` or `other` but not both. $(s \Delta s_2 \stackrel{\text{def}}{=} (s \cup s_2) \setminus (s_2 \cap s))$

Math. operations on sets: union, intersection etc

A complicated syntax ?

Syntax is natural here again: `|`, `&`, `-`, $\wedge$.

4 operations coming from maths ($\cup, \cap, \setminus, \Delta$) *(new!)*

Create new set with elements. . . :

- `s.union(other, ...)` or `s | other | ...`
  from the set `s` and all `others`. $(s \cup s_2 \cup \dots)$
- `s.intersection(other, ...)` or `s & other & ...`
  common to the set `s` and all `others`. $(s \cap s_2 \cap \dots)$
- `s.difference(other, ...)` or `s - other - ...`
  in the set `s` that are not in the `others`. $((s \setminus s_2) \setminus s_3 \dots))$
- `s.symmetric_difference(other)` or `s ^ other` *((new!))*
  in either the set `s` or `other` but not both. $(s \Delta s_2 \stackrel{\text{def}}{=} (s \cup s_2) \setminus (s_2 \cap s))$

Math. operations on sets: union, intersection etc

A complicated syntax ?

Syntax is natural here again: `|`, `&`, `-`, $\wedge$.

4 operations coming from maths ($\cup, \cap, \setminus, \Delta$) *(new!)*

Create new set with elements. . . :

- `s.union(other, ...)` or `s | other | ...`
  from the set `s` and all `others`. $(s \cup s_2 \cup \dots)$
- `s.intersection(other, ...)` or `s & other & ...`
  common to the set `s` and all `others`. $(s \cap s_2 \cap \dots)$
- `s.difference(other, ...)` or `s - other - ...`
  in the set `s` that are not in the `others`. $((s \setminus s_2) \setminus s_3 \dots))$
- `s.symmetric_difference(other)` or `s ^ other` *((new!))*
  in either the set `s` or `other` but not both. $(s \Delta s_2 \stackrel{\text{def}}{=} (s \cup s_2) \setminus (s_2 \cap s))$

Math. operations on sets: union, intersection etc

A complicated syntax ?

Syntax is natural here again: `|`, `&`, `-`, $\wedge$.

4 operations coming from maths ($\cup, \cap, \setminus, \Delta$) *(new!)*

Create new set with elements. . . :

- `s.union(other, ...)` or `s | other | ...`
  from the set `s` and all `others`. $(s \cup s_2 \cup \dots)$
- `s.intersection(other, ...)` or `s & other & ...`
  common to the set `s` and all `others`. $(s \cap s_2 \cap \dots)$
- `s.difference(other, ...)` or `s - other - ...`
  in the set `s` that are not in the `others`. $((s \setminus s_2) \setminus s_3 \dots))$
- `s.symmetric_difference(other)` or `s ∧ other` *((new!))*
  in either the set `s` or `other` but not both. $(s \Delta s_2 \stackrel{\text{def}}{=} (s \cup s_2) \setminus (s_2 \cap s))$

Math. operations on sets: union, intersection etc

A complicated syntax ?

Syntax is natural here again: `|`, `&`, `-`, $\wedge$.

4 operations coming from maths ($\cup, \cap, \setminus, \Delta$) *(new!)*

Create new set with elements. . . :

- `s.union(other, ...)` or `s | other | ...`
  from the set `s` and all `others`. $(s \cup s_2 \cup \dots)$
- `s.intersection(other, ...)` or `s & other & ...`
  common to the set `s` and all `others`. $(s \cap s_2 \cap \dots)$
- `s.difference(other, ...)` or `s - other - ...`
  in the set `s` that are not in the `others`. $((s \setminus s_2) \setminus s_3 \dots))$
- `s.symmetric_difference(other)` or `s ^ other` *((new!))*
  in either the set `s` or `other` but not both. $(s \Delta s_2 \stackrel{\text{def}}{=} (s \cup s_2) \setminus (s_2 \cap s))$

Modifying a set: union, intersection etc

5 **methods** for sets: *(new!)*

Create new set with elements. . . :

- `s.add(elem)` Adds element `elem` to the set `s`.
  Does not change `s` if `elem` was already present!
- `s.remove(elem)` Removes element `elem` from the set `s`.
  Raises KeyError if `elem` is *not contained* in the set.
- `s.discard(elem)` Removes element `elem` from the set `s` if it is present.
- `s.pop()` Removes and returns an **arbitrary** element from the set `s`.
  Raises KeyError for *an empty set*. Arbitrary? Yes, there is no order!
- `s.clear()` Removes **all elements** from the set `s` (like `s = set()`).

Getting a **fresh** copy of a set? *(new!)*

`s.copy()` returns a new set with a **fresh** copy of the set `s`.

Modifying a set: union, intersection etc

5 **methods** for sets: *(new!)*

Create new set with elements. . . :

- `s.add(elem)` Adds element `elem` to the set `s`.
  Does not change `s` if `elem` was already present!
- `s.remove(elem)` Removes element `elem` from the set `s`.
  Raises KeyError if `elem` is *not contained* in the set.
- `s.discard(elem)` Removes element `elem` from the set `s` if it is present.
- `s.pop()` Removes and returns an **arbitrary** element from the set `s`.
  Raises KeyError for *an empty set*. Arbitrary? Yes, there is no order!
- `s.clear()` Removes **all elements** from the set `s` (like `s = set()`).

Getting a **fresh** copy of a set? *(new!)*

`s.copy()` returns a new set with a **fresh** copy of the set `s`.

Modifying a set: union, intersection etc

5 **methods** for sets: *(new!)*

Create new set with elements. . . :

- `s.add(elem)` Adds element `elem` to the set `s`.
  Does not change `s` if `elem` was already present!
- `s.remove(elem)` Removes element `elem` from the set `s`.
  Raises KeyError if `elem` is *not contained* in the set.
- `s.discard(elem)` Removes element `elem` from the set `s` if it is present.
- `s.pop()` Removes and returns an **arbitrary** element from the set `s`.
  Raises KeyError for *an empty set*. Arbitrary? Yes, there is no order!
- `s.clear()` Removes **all elements** from the set `s` (like `s = set()`).

Getting a **fresh** copy of a set? *(new!)*

`s.copy()` returns a new set with a **fresh** copy of the set `s`.

Modifying a set: union, intersection etc

5 **methods** for sets: *(new!)*

Create new set with elements. . . :

- `s.add(elem)` Adds element `elem` to the set `s`.
  Does not change `s` if `elem` was already present!
- `s.remove(elem)` Removes element `elem` from the set `s`.
  Raises KeyError if `elem` is *not contained* in the set.
- `s.discard(elem)` Removes element `elem` from the set `s` if it is present.
- `s.pop()` Removes and returns an **arbitrary** element from the set `s`.
  Raises KeyError for *an empty set*. Arbitrary? Yes, there is no order!
- `s.clear()` Removes **all elements** from the set `s` (like `s = set()`).

Getting a **fresh** copy of a set? *(new!)*

`s.copy()` returns a new set with a **fresh** copy of the set `s`.

Modifying a set: union, intersection etc

5 **methods** for sets: *(new!)*

Create new set with elements. . . :

- `s.add(elem)` Adds element `elem` to the set `s`.
  Does not change `s` if `elem` was already present!
- `s.remove(elem)` Removes element `elem` from the set `s`.
  Raises KeyError if `elem` is *not contained* in the set.
- `s.discard(elem)` Removes element `elem` from the set `s` if it is present.
- `s.pop()` Removes and returns an **arbitrary** element from the set `s`.
  Raises KeyError for *an empty set*. Arbitrary? Yes, there is no order!
- `s.clear()` Removes **all elements** from the set `s` (like `s = set()`).

Getting a **fresh** copy of a set? *(new!)*

`s.copy()` returns a new set with a **fresh** copy of the set `s`.

Modifying a set: union, intersection etc

5 **methods** for sets: *(new!)*

Create new set with elements. . . :

- `s.add(elem)` Adds element `elem` to the set `s`.
  Does not change `s` if `elem` was already present!
- `s.remove(elem)` Removes element `elem` from the set `s`.
  Raises KeyError if `elem` is *not contained* in the set.
- `s.discard(elem)` Removes element `elem` from the set `s` if it is present.
- `s.pop()` Removes and returns an **arbitrary** element from the set `s`.
  Raises KeyError for *an empty set*. Arbitrary? Yes, there is no order!
- `s.clear()` Removes **all elements** from the set `s` (like `s = set()`).

Getting a **fresh** copy of a set? *(new!)*

`s.copy()` returns a new set with a **fresh** copy of the set `s`.

Sum-up about **sets**

About sets, we saw: *(new!)*

- concept of a set (from mathematics),
- how to define it in Python, read or modify it or its elements,
- some mathematical operations on sets (union, intersection etc),
- methods for modifying sets (`add`, `remove`, `discard`, etc)
- **set comprehension** can be used **exactly** as list comprehension:

```
s = { (-1)**a + b/a for a in range(1,100) for b in range(-10,10) } .
```

- last remark: a **frozenset** is an **unmutable** set.

Thanks for listening to this part about *lists* and *sets*

Any question?

Reference websites

- www.MahindraEcoleCentrale.edu.in/portal : **MEC Moodle**,
- the IntroToPython.org website,
- and the Python documentation at docs.python.org/2/,

Want to know more?

↪ practice (by yourself) with these websites!

↪ and contact me (e-mail, *flying pigeons*, Moodle etc) if needed.

Next week will be about **dictionaries** in Python!

Like `verbs = { 'avoir' : 'to have', 'être' : 'to be', ... }`

Outline for today (16th March) Lecture 3/3

1. Presentation
2. A quick and interactive sum-up of the first two months
3. Data structures for programming cleverly (3 lectures)
   - Lists in Python (1/3)
   - Sets in Python (2/3)
   - Dictionaries in Python (3/3)
4. Sum-um about data structures : lists, sets, and dicts

Definition and concept of a dictionary in Python

Concept of a dictionnary and definition *(new!)*

"A dictionary is an associative table, ie. an ***unordered*** *collection of* ***pairs*** `key : value`*."*

Goal ?

Easily store **connected bits of information**.
For example, you might store your name, section and roll number together:
`hero = {'name' : 'Luke S.', 'section' : 'A1', 'roll' : 438}`.

Sets and dicts have a similar limitation

The keys should be **hashable**, so only one value can be associated with a key.
As for elements of a set, **keys of a dict need to have a `__hash__` method**.

Definition and concept of a dictionary in Python

Concept of a dictionnary and definition *(new!)*

"A dictionary is an associative table, ie. an ***unordered*** *collection of* ***pairs*** `key : value`*."*

Goal ?

Easily store **connected bits of information**.
For example, you might store your name, section and roll number together:
`hero = {'name' : 'Luke S.', 'section' : 'A1', 'roll' : 438}`.

Sets and dicts have a similar limitation

The keys should be **hashable**, so only one value can be associated with a key.
As for elements of a set, **keys of a dict need to have a `__hash__` method.**

Definition and concept of a dictionary in Python

Concept of a dictionnary and definition *(new!)*

"A dictionary is an associative table, ie. an ***unordered*** *collection of* ***pairs*** `key : value`*."*

Goal ?

Easily store **connected bits of information**.
For example, you might store your name, section and roll number together:
`hero = {'name' : 'Luke S.', 'section' : 'A1', 'roll' : 438}`.

Sets and dicts have a similar limitation

The keys should be **hashable**, so only one value can be associated with a key.
As for elements of a set, **keys of a dict need to have a `__hash__` method**.

Two first examples of dictionaries in Python

Description of courses?

Associating an adjective to the name of a course:

```
d = {'CS101' : 'awesome', 'MA102' : 'good', 'CB101' : 'cooking',
                          'HS103' : 'tiring!'}
```

A dictionary is like a *generalized list*, for which the keys were just `0`, `1`, ..., `n-1` (and the order matters).
The order **does not matter** for a dictionary!

Some French verbs and their definition (like a real dictionary)

```
verbs = { 'avoir' : 'to have', 'être' : 'to be', ...  }
```

Online reference (for practicing)

⟶ introtopython.org/dictionaries.html ⟵

Two first examples of dictionaries in Python

Description of courses?

Associating an adjective to the name of a course:

```
d = {'CS101' : 'awesome', 'MA102' : 'good', 'CB101' : 'cooking',
                          'HS103' : 'tiring!'}
```

A dictionary is like a *generalized list*, for which the keys were just `0`, `1`, ..., `n-1` (and the order matters).
The order **does not matter** for a dictionary!

Some French verbs and their definition (like a real dictionary)

```
verbs = { 'avoir' : 'to have', 'être' : 'to be', ...  }
```

Online reference (for practicing)

⟶ introtopython.org/dictionaries.html ⟵

Two first examples of dictionaries in Python

Description of courses?

Associating an adjective to the name of a course:

```
d = {'CS101' : 'awesome', 'MA102' : 'good', 'CB101' : 'cooking',
                          'HS103' : 'tiring!'}
```

A dictionary is like a *generalized list*, for which the keys were just `0`, `1`, ..., `n-1` (and the order matters).
The order **does not matter** for a dictionary!

Some French verbs and their definition (like a real dictionary)

```
verbs = { 'avoir' : 'to have', 'être' : 'to be', ...  }
```

Online reference (for practicing)

⟶ introtopython.org/dictionaries.html ⟵

Basic notions about dicts in Python

Size or length of a dict *(new!)*

- `len(d)` : **size** of the dict `d` (ie. number of pairs `key : value`).

Reading the **value** attached to a **key** *(new!)*

Same syntax for **lists** and **dicts**, but indexing is done with the **keys**:

- `d[key]`: try to read the value paired with the key.
  Raises KeyError if the `key` is not used in the dictionary.
- `d.has_key(key)` or `key in d`: `True` if the `key` is used, `False` otherwise.

Simple examples (*Question:* what is `d[k]` here?)

If `d = {0:[], 1:[1], 5:[1, 5], 12:[1, 2, 3, 4, 6, 12]}`:

- `len(d)` is ?
- `d[0]` is ?, `d[2]` is ?, `d[5]` is ?.
- `0 in d` is ?, `2 in d` is ?, `12 not in d` is ?, `5 not in d` is ?.
- `d[k]` can also be **modified**: `d[0] = 42` will change the dictionary `d`.

Basic notions about dicts in Python

Size or length of a dict *(new!)*

- `len(d)` : **size** of the dict `d` (ie. number of pairs `key : value`).

Reading the **value** attached to a **key** *(new!)*

Same syntax for **lists** and **dicts**, but indexing is done with the **keys**:

- `d[key]`: try to read the value paired with the key.
  Raises KeyError if the `key` is not used in the dictionary.
- `d.has_key(key)` or `key in d`: `True` if the `key` is used, `False` otherwise.

Simple examples (*Question:* what is `d[k]` here?)

If `d = {0:[], 1:[1], 5:[1, 5], 12:[1, 2, 3, 4, 6, 12]}`:

- `len(d)` is ?
- `d[0]` is ?, `d[2]` is ?, `d[5]` is ?.
- `0 in d` is ?, `2 in d` is ?, `12 not in d` is ?, `5 not in d` is ?.
- `d[k]` can also be modified: `d[0] = 42` will change the dictionary `d`.

Basic notions about dicts in Python

Size or length of a dict *(new!)*

- `len(d)` : **size** of the dict `d` (ie. number of pairs `key : value`).

Reading the **value** attached to a **key** *(new!)*

Same syntax for **lists** and **dicts**, but indexing is done with the **keys**:

- `d[key]`: try to read the value paired with the key.
  Raises KeyError if the `key` is not used in the dictionary.
- `d.has_key(key)` or `key in d`: `True` if the `key` is used, `False` otherwise.

Simple examples (*Question:* what is `d[k]` here?)

If `d = {0:[], 1:[1], 5:[1, 5], 12:[1, 2, 3, 4, 6, 12]}`:

- `len(d)` is ?
- `d[0]` is ?, `d[2]` is ?, `d[5]` is ?.
- `0 in d` is ?, `2 in d` is ?, `12 not in d` is ?, `5 not in d` is ?.
- `d[k]` can also be **modified**: `d[0] = 42` will change the dictionary `d`.

Basic notions about dicts in Python

Size or length of a dict *(new!)*

- `len(d)` : **size** of the dict `d` (ie. number of pairs `key : value`).

Reading the **value** attached to a **key** *(new!)*

Same syntax for **lists** and **dicts**, but indexing is done with the **keys**:

- `d[key]`: try to read the value paired with the key.
  Raises KeyError if the `key` is not used in the dictionary.
- `d.has_key(key)` or `key in d`: `True` if the `key` is used, `False` otherwise.

Simple examples (*Question:* what is `d[k]` here?)

If `d = {0:[], 1:[1], 5:[1, 5], 12:[1, 2, 3, 4, 6, 12]}`:

- `len(d)` is ?
- `d[0]` is ?, `d[2]` is ?, `d[5]` is ?.
- `0 in d` is ?, `2 in d` is ?, `12 not in d` is ?, `5 not in d` is ?.
- `d[k]` can also be **modified**: `d[0] = 42` will change the dictionary `d`.

One "real-world" example of a dictionary

Demo of a *Windows* .exe program written in Python

This demo is a fun way to browse through your photos (from *your* ID cards).
It uses a database of your batch, written internally as a **Python dictionary**.
Keys are roll numbers, and each value is itself a dictionary (nested dicts!):

Example with the first student of the list:

```
students = {
    "14XJ00000" : {
        "name" : "Brad Pitt",
        "branch" : "ME",
        "counsellor" : "Edward Norton",
        "emailaddress" : "b.pitt@fight.club",
        "group" : "A0",
        "rollnumber" : "14XJ00000"
    },
    # ... there is 230 more students like this!
}
```

One "real-world" example of a dictionary

Demo of a *Windows* .exe program written in Python

This demo is a fun way to browse through your photos (from *your* ID cards).
It uses a database of your batch, written internally as a **Python dictionary**.
Keys are roll numbers, and each value is itself a dictionary (nested dicts!):

Example with the first student of the list:

```
students = {
    "14XJ00000" : {
        "name" : "Brad Pitt",
        "branch" : "ME",
        "counsellor" : "Edward Norton",
        "emailaddress" : "b.pitt@fight.club",
        "group" : "A0",
        "rollnumber" : "14XJ00000"
    },
    # ... there is 230 more students like this!
}
```

Several approaches to create a dict

Python offers different syntaxes for creating a dictionary.
To illustrate, the following examples all return a dictionary equal to
`{'one' : 1, 'two' : 2, 'three' : 3}`

4 examples *(new!)*

- `a = {'two' : 2, 'one' : 1, 'three' : 3}` is *the best syntax*,
- `b = dict(one=1, two=2, three=3)` uses `dict` as a function with keyword arguments (argument `one` has value `1`, `two` has value `2` etc),
- `c = dict([('two', 2), ('one', 1), ('three', 3)])` from a list of pairs `(key, value)`,
- `d = dict(zip(['one', 'two', 'three'], [1, 2, 3]))` with the `zip` function.

Four way to define a dict?

With the above 4 commands, `a == b == c == d` is `True`.

Several approaches to create a dict

Python offers different syntaxes for creating a dictionary.
To illustrate, the following examples all return a dictionary equal to
`{'one' : 1, 'two' : 2, 'three' : 3}`

4 examples *(new!)*

- `a = {'two' : 2, 'one' : 1, 'three' : 3}` is *the best syntax*,
- `b = dict(one=1, two=2, three=3)` uses `dict` as a function with **keyword arguments** (argument `one` has value `1`, `two` has value `2` etc),
- `c = dict([('two', 2), ('one', 1), ('three', 3)])` from a list of pairs `(key, value)`,
- `d = dict(zip(['one', 'two', 'three'], [1, 2, 3]))` with the `zip` function.

Four way to define a dict?

With the above 4 commands, `a == b == c == d` is `True`.

Several approaches to create a dict

Python offers different syntaxes for creating a dictionary.
To illustrate, the following examples all return a dictionary equal to
`{'one' : 1, 'two' : 2, 'three' : 3}`

4 examples *(new!)*

- `a = {'two' : 2, 'one' : 1, 'three' : 3}` is *the best syntax*,
- `b = dict(one=1, two=2, three=3)` uses `dict` as a function with **keyword arguments** (argument `one` has value `1`, `two` has value `2` etc),
- `c = dict([('two', 2), ('one', 1), ('three', 3)])` from a list of pairs `(key, value)`,
- `d = dict(zip(['one', 'two', 'three'], [1, 2, 3]))` with the `zip` function.

Four way to define a dict?

With the above 4 commands, `a == b == c == d` is `True`.

Several approaches to create a dict

Python offers different syntaxes for creating a dictionary.
To illustrate, the following examples all return a dictionary equal to
`{'one' : 1, 'two' : 2, 'three' : 3}`

4 examples *(new!)*

- `a = {'two' : 2, 'one' : 1, 'three' : 3}` is *the best syntax*,
- `b = dict(one=1, two=2, three=3)` uses `dict` as a function with **keyword arguments** (argument `one` has value `1`, `two` has value `2` etc),
- `c = dict([('two', 2), ('one', 1), ('three', 3)])` from a list of pairs `(key, value)`,
- `d = dict(zip(['one', 'two', 'three'], [1, 2, 3]))` with the `zip` function.

Four way to define a dict?

With the above 4 commands, `a == b == c == d` is `True`.

Several approaches to create a dict

Python offers different syntaxes for creating a dictionary.
To illustrate, the following examples all return a dictionary equal to
`{'one' : 1, 'two' : 2, 'three' : 3}`

4 examples *(new!)*

- `a = {'two' : 2, 'one' : 1, 'three' : 3}` is *the best syntax*,
- `b = dict(one=1, two=2, three=3)` uses `dict` as a function with **keyword arguments** (argument `one` has value `1`, `two` has value `2` etc),
- `c = dict([('two', 2), ('one', 1), ('three', 3)])` from a list of pairs `(key, value)`,
- `d = dict(zip(['one', 'two', 'three'], [1, 2, 3]))` with the `zip` function.

Four way to define a dict?

With the above 4 commands, `a == b == c == d` is `True`.

Looping over dicts

An easy syntax !

Syntax is natural, similar to `for` loops with lists:

```
for k in mydict:
    # something here can be done
    # with k as the key, mydict[k] as the value
```

3 ways of looping over a dictionary *(new!)*

- looping over the `keys`: `for k in mydict`.
  The key `k` can be used, and `mydict[k]` can be used or modified.
- looping over the `values`: `for v in mydict.values()`.
  The value `v` can be used, but **modifying `v` does not modify `mydict`!**
- looping over the pairs of `(key, value)`: `for k, v in mydict.items()`.
  This is the more general method.

Looping over dicts

An easy syntax !

Syntax is natural, similar to `for` loops with lists:

```
for k in mydict:
    # something here can be done
    # with k as the key, mydict[k] as the value
```

3 ways of looping over a dictionary *(new!)*

- looping over the `keys`: `for k in mydict`.
  The key `k` can be used, and `mydict[k]` can be used or modified.
- looping over the `values`: `for v in mydict.values()`.
  The value `v` can be used, but **modifying `v` does not modify `mydict`!**
- looping over the pairs of `(key, value)`: `for k, v in mydict.items()`.
  This is the more general method.

Looping over dicts

An easy syntax !

Syntax is natural, similar to `for` loops with lists:

```
for k in mydict:
    # something here can be done
    # with k as the key, mydict[k] as the value
```

3 ways of looping over a dictionary *(new!)*

- looping over the `keys`: `for k in mydict`.
  The key `k` can be used, and `mydict[k]` can be used or modified.
- looping over the `values`: `for v in mydict.values()`.
  The value `v` can be used, but **modifying `v` does not modify `mydict`!**
- looping over the pairs of `(key, value)`: `for k, v in mydict.items()`.
  This is the more general method.

Looping over dicts

An easy syntax !

Syntax is natural, similar to `for` loops with lists:

```
for k in mydict:
    # something here can be done
    # with k as the key, mydict[k] as the value
```

3 ways of looping over a dictionary *(new!)*

- looping over the `keys`: `for k in mydict`.
  The key `k` can be used, and `mydict[k]` can be used or modified.
- looping over the `values`: `for v in mydict.values()`.
  The value `v` can be used, but **modifying `v` does not modify `mydict`!**
- looping over the pairs of `(key, value)`: `for k, v in mydict.items()`.
  This is the more general method.

Removing keys or values from a dictionary

One trick and 3 **methods** for dicts: *(new!)*

Remove values or keys from a dict:

- `d.pop(k)` removes and returns the value associated with they `key` `k`.
  Raises KeyError if *the key is not in the dict.*
- same as `del d[k]`: the `del` command is used to delete the value `d[k]`.
- `d.popitem()` removes and returns an **arbitrary** pair `key : value` from the dict `d`.
  Raises KeyError for *an empty dict.* Arbitrary? Yes, there is no order!
- `d.clear()` removes **all elements** from the dict `d` (like doing `d = {}`).

Getting a **fresh** copy of a dict? *(new!)*

`d.copy()` returns a new dict with a **fresh** copy of the dict `d`.

Removing keys or values from a dictionary

One trick and 3 **methods** for dicts: *(new!)*

Remove values or keys from a dict:

- `d.pop(k)` removes and returns the value associated with they `key` k.
  Raises KeyError if *the key is not in the dict.*
- same as `del d[k]`: the `del` command is used to delete the value `d[k]`.
- `d.popitem()` removes and returns an **arbitrary** pair `key : value` from the dict `d`.
  Raises KeyError for *an empty dict.* Arbitrary? Yes, there is no order!
- `d.clear()` removes **all elements** from the dict `d` (like doing `d = {}`).

Getting a **fresh** copy of a dict? *(new!)*

`d.copy()` returns a new dict with a **fresh** copy of the dict `d`.

Removing keys or values from a dictionary

One trick and 3 **methods** for dicts: *(new!)*

Remove values or keys from a dict:

- `d.pop(k)` removes and returns the value associated with they `key` k.
  Raises KeyError if *the key is not in the dict.*
- same as `del d[k]`: the `del` command is used to delete the value `d[k]`.
- `d.popitem()` removes and returns an **arbitrary** pair `key : value` from the dict `d`.
  Raises KeyError for *an empty dict.* Arbitrary? Yes, there is no order!
- `d.clear()` removes **all elements** from the dict `d` (like doing `d = {}`).

Getting a **fresh** copy of a dict? *(new!)*

`d.copy()` returns a new dict with a **fresh** copy of the dict `d`.

Removing keys or values from a dictionary

One trick and 3 **methods** for dicts: *(new!)*

Remove values or keys from a dict:

- `d.pop(k)` removes and returns the value associated with they `key` k.
  Raises KeyError if *the key is not in the dict.*
- same as `del d[k]`: the `del` command is used to delete the value `d[k]`.
- `d.popitem()` removes and returns an **arbitrary** pair `key : value` from the dict `d`.
  Raises KeyError for *an empty dict.* Arbitrary? Yes, there is no order!
- `d.clear()` removes **all elements** from the dict `d` (like doing `d = {}`).

Getting a **fresh** copy of a dict? *(new!)*

`d.copy()` returns a new dict with a **fresh** copy of the dict `d`.

Converting these 3 data structures with each other

As always in Python, it is easy to convert[1] values to other data-types.

Creating a list... with the function `list`!

The function `list` can be *applied* to `tuples`, `strings`, `sets` and `dicts`.

Creating a set... with the function `set`!

The function `set` can be *applied* to `lists`, `tuples`, `strings` and `dicts`.

Creating a dict... with the function `dict`!

The function `dict` can be *applied* to `lists`, `tuples` or `sets` of pairs `(key, value)`.

Examples of each functions

Examples: on the IPython console.
I will try to give examples with list, sets and dicts comprehension also!

[1] You will see soon that in fact we operate on an object, instance of a class, changed to be instance of another class.

Converting these 3 data structures with each other

As always in Python, it is easy to convert[1] values to other data-types.

Creating a list... with the function `list`!

The function `list` can be *applied* to `tuples`, `strings`, `sets` and `dicts`.

Creating a set... with the function `set`!

The function `set` can be *applied* to `lists`, `tuples`, `strings` and `dicts`.

Creating a dict... with the function `dict`!

The function `dict` can be *applied* to `lists`, `tuples` or `sets` of pairs `(key, value)`.

Examples of each functions

Examples: on the IPython console.
I will try to give examples with list, sets and dicts comprehension also!

[1] You will see soon that in fact we operate on an object, instance of a class, changed to be instance of another class.

Converting these 3 data structures with each other

As always in Python, it is easy to convert[1] values to other data-types.

Creating a list... with the function `list`!

The function `list` can be *applied* to `tuples`, `strings`, `sets` and `dicts`.

Creating a set... with the function `set`!

The function `set` can be *applied* to `lists`, `tuples`, `strings` and `dicts`.

Creating a dict... with the function `dict`!

The function `dict` can be *applied* to `lists`, `tuples` or `sets` of pairs `(key, value)`.

Examples of each functions

Examples: on the IPython console.
I will try to give examples with list, sets and dicts comprehension also!

[1] You will see soon that in fact we operate on an object, instance of a class, changed to be instance of another class.

Converting these 3 data structures with each other

As always in Python, it is easy to convert[1] values to other data-types.

Creating a list... with the function `list`!

The function `list` can be *applied* to `tuples`, `strings`, `sets` and `dicts`.

Creating a set... with the function `set`!

The function `set` can be *applied* to `lists`, `tuples`, `strings` and `dicts`.

Creating a dict... with the function `dict`!

The function `dict` can be *applied* to `lists`, `tuples` or `sets` of pairs `(key, value)`.

Examples of each functions

Examples: on the IPython console.
I will try to give examples with list, sets and dicts comprehension also!

[1] You will see soon that in fact we operate on an object, instance of a class, changed to be instance of another class.

Sum-up about **dicts**

About dicts, we saw: *(new!)*

- concept of a dictionary (it is an *associative array*),
- how to define it in Python, and basically read or modify it or its elements,
- 3 different ways of looping over a dict: loop over the keys, or the values, or both,
- it is easy to convert between lists, sets, and dicts,
- **dict comprehension** can be used **exactly** as list comprehension:
  `roots = { i**3 : i for i in xrange(4) }` ,
- there is more methods for modifying dicts (`get`, `update`, `setdefault`, etc)
- last remark: there is *no* **unmutable** counterpart of a dict (like tuples for lists or frozensets for sets).

Quick reminders (if we have time)

Lists (and tuples) (1/3) — Ordered collection of any objects

- one last small example `mylist = [1, 3, 5]`, `mytuple = (2, 6)`,
- a list is **mutable** (`mylist[0] = 3` is OK), a tuple is **not**,
- qualities: simple to use, many functions/methods, easy to loop or modify.

Sets (2/3) — Unordered collection of *unique* elements

- one last small example `myset = {-1, 1}`,
- qualities: efficiently deal with **finite** sets (especially with *set comprehension*), operations on sets are exactly coming from maths.

Dictionaries (3/3) — Unordered collection of pairs of *unique* (`key : value`)

- one last small example
  `wines = {'red' : 'tasty', 'white' : 'sweet', 'champagne' : 'party time!'}`,
- qualities: simple to store connected bits of information, easy to manipulate and loop over.

Quick reminders (if we have time)

Lists (and tuples) (1/3) — Ordered collection of any objects

- one last small example `mylist = [1, 3, 5]`, `mytuple = (2, 6)`,
- a list is **mutable** (`mylist[0] = 3` is OK), a tuple is **not**,
- qualities: simple to use, many functions/methods, easy to loop or modify.

Sets (2/3) — Unordered collection of *unique* elements

- one last small example `myset = {-1, 1}`,
- qualities: efficiently deal with **finite** sets (especially with *set comprehension*), operations on sets are exactly coming from maths.

Dictionaries (3/3) — Unordered collection of pairs of *unique* (`key : value`)

- one last small example
  `wines = {'red' : 'tasty', 'white' : 'sweet', 'champagne' : 'party time!'},`
- qualities: simple to store connected bits of information, easy to manipulate and loop over.

Quick reminders (if we have time)

Lists (and tuples) (1/3) — Ordered collection of any objects

- one last small example `mylist = [1, 3, 5]`, `mytuple = (2, 6)`,
- a list is **mutable** (`mylist[0] = 3` is OK), a tuple is **not**,
- qualities: simple to use, many functions/methods, easy to loop or modify.

Sets (2/3) — Unordered collection of *unique* elements

- one last small example `myset = {-1, 1}`,
- qualities: efficiently deal with **finite** sets (especially with *set comprehension*), operations on sets are exactly coming from maths.

Dictionaries (3/3) — Unordered collection of pairs of *unique* (`key : value`)

- one last small example
```
wines = {'red' : 'tasty', 'white' : 'sweet',
'champagne' : 'party time!'},
```
- qualities: simple to store connected bits of information, easy to manipulate and loop over.

Thanks for listening to this lecture about dictionaries

Any question?

Reference websites

- www.MahindraEcoleCentrale.edu.in/portal : **MEC Moodle**,
- the IntroToPython.org website,
- and the Python documentation at docs.python.org/2/,

Want to know more?

↪ practice by yourself with these websites !

↪ and contact me (e-mail, *flying pigeons*, Moodle etc) if needed

↪ or read the *"Python in Easy Steps"* book on Python, or the *"Introduction to Algorithms"* (Cormen) for *concepts of data structures.*

Next lectures: **modules**, **OOP** and **exceptions** by Prof. Vipin