In [1]:
```
from jyquickhelper import add_notebook_menu
add_notebook_menu(first_level=1, last_level=4, header="<font color='blus'>Iterators and
```

Out[1]: **Iterators and generators**

run previous cell, wait for 2 seconds

Iterators

> An iterator makes it possible to manipulate large amounts of data without having to store them in memory.
>
> These data can be processed one by one and not necessarily in batch.

> How to recognize a `Python` iterator ?
>
> `list` is an iterator.

In [39]:
```
'__iter__' in dir(list)
```

Out[39]:
```
True
```

In [46]:
```
'__iter__' in dir(tuple)
```

Out[46]:
```
True
```

Generators : `yield`

> A generator is used in a loop like `for` or `while`. It produces a value for each one of the iteration.
>
> Using in a function, `yield` returns a generator.
>
> Generators can be paused then resumed.

In [40]:
```
def int_generator(n):
    '''Integer generator.
    '''
    i = 0
    while i < n:
        yield i # each call to yield produces a value
        i += 1
```

> Then data from generator can be consumed

In [5]:
```
list_int = list()
for i in int_generator(10):
    list_int.append(i)
print(list_int)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

> Once a generator is empty, you have to recreate one.

In [8]:
```
generator = int_generator(10)# Create a new generator
```

In [9]:
```
list_int = list()
for i in generator :
    list_int.append(i)

print(list_int)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In [10]:
```
# Lets show that generator is empty
list_int = list()
for i in generator :
    list_int.append(i)

print(list_int)
```

```
[]
```

> Building of a generator in comprehension

In [11]:
```
generator2 = (i for i in int_generator(20))
```

In [12]:
```
# Mise en évidence de l'épuisement du générateur
list_int = list()
for i in generator2 :
    list_int.append(i)

print(list_int)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

> Retrieve the index and the value: `enumerate`

In [56]:
```
generator2 = (-i for i in int_generator(20))
```

In [13]:
```
for i, val in enumerate(generator2) :
    print(i, val)
```

In [52]:
```
def mygenerator():
    n = 1
    yield n
    n += 2
    yield n

print(mygenerator())
for i in mygenerator() :
    print(i)
    if i == 10 :
        break
```

```
<generator object mygenerator at 0x7f446d87de40>
1
3
```

Randomness

Generating random values

random.random

> Generating a aandom number between 0 and 1

In [14]:
```python
import random
print(random.random())
print(random.random())
```

```
0.24615502986656357
0.9652104833337973
```

> Successive calls to `random.random` return different random values.

> The `seed` method allows to generate the same random values for successive calls

In [15]:
```python
random.seed(10)
list_random = [random.random() for _ in range(3)]
print(list_random)
```

```
[0.5714025946899135, 0.4288890546751146, 0.5780913011344704]
```

In [16]:
```python
random.seed(10)
list_random = [random.random() for _ in range(3)]
print(list_random)
```

```
[0.5714025946899135, 0.4288890546751146, 0.5780913011344704]
```

random.range

> Generating random values within a range of values.

In [26]:
```python
# Generating a random number between 3 and 5, upper limit is excluded
print(random.randrange(3, 6))

# Generating a random number between 3 and 6
list_ = [random.randrange(3, 6) for i in range(10)]
print(list_)

# Generating a list of random numbers between 0 and 9
list_ = [random.randrange(10) for i in range(10)]
print(list_)
```

```
5
[5, 3, 4, 3, 5, 4, 3, 4, 5, 4]
[3, 5, 2, 1, 9, 3, 6, 3, 1, 4]
```

Selecting random values

random.choice

In [ ]:
```python
list_color = ['green', 'blue', 'pink', 'yellow', 'red']
print(list_color)
```

> We randomly choose an element in the list

In [71]:
```python
print(random.choice(list_color))
```

```
yellow
```

random.sample

> Generation of a random sample
>
> Below, the choice relates to a sample of size 1 in the list

In [69]:
```python
print(random.sample(list_color,1))
```

```
['yellow']
```

random.shuffle

> Shuffles the list passed as an argument and returns `None`.

In [80]:
```python
print(list_color)
random.shuffle(list_color)
print(list_color)
```

```
['green', 'red', 'blue', 'pink', 'yellow']
['red', 'pink', 'yellow', 'green', 'blue']
```

Regular expressions

- https://python.doctor/page-expressions-regulieres-regular-python

> These expressions are used to detect a sequence of characters, mean, an expression, in another character sequence.

Interpreted characters in regulars expressions

> Interpreting a character means that character matches with a specific action.
>
> These characters are exclusively:
>
> > . ^ $ * + ? { } [ ] \ | ( )

These characters are interpreted as follows in regular expressions:

- `.` The dot matches any character.
- `^` Indicates a beginning of a sequence but also means "opposite of"
- `$` End of sequence
- `[xy]` A list of possible characters. Example `[abc]` is equivalent to: `a` or `b` or `c`

  ```
  [0-9] indicates that the list of possible characters is 0, 1, ..., 9
  ```
- `(x|y)` Indicates a typical multiple choice (alfa|beta) is equivalent to "alfa" OR "beta"
- `\d` the sequence consists only of digits, which is equivalent to [0-9].
- `\D` the segment is not composed of digits, which is equivalent to [^0-9].
- `\s` A space, which is equivalent to [\t\n\r\f\v].
- `\S` No space, which is equivalent to [^ \t\n\r\f\v].
- `\w` Alphanumeric presence, equivalent to [a-zA-Z0-9_].
- `\W` No alphanumeric presence [^a-zA-Z0-9_].
- `\` Is an escape character that allows any of the characters `. ^` `$` * + ? { } [ ] \ | ( )` to be processedas a normal character.
- `D{2}` indicates two occurrences of the character D
- `REP{1,9}` : the REP sequence can be repeated from 1 to 9 times.
- `REP{0,9}` : the REP sequence can be absent or repeated up to 9 times.
- `REP{1,}` : the REP sequence is repeated at least once.
- `(.)?` : 0 or 1 character expected in expression (any character)
- `(.)+` : at least one character expected in the expression (any character(s))
- `(.)*` : zero or more characters expected in the expression (any characters)

Matching an expression with a sequence: `re.match`

In [27]:
```
import re
```

> Checking of the presence of a sequence of characters in another sequence of characters

In [29]:
```
# Is the expression composed of any characters from A to Z and ending with S character p
expression = '[A-Z]+S$'
sequence = 'DATASCIENCES'
res_match = re.match(expression, sequence)
print('match' in str(type(res_match)).lower())
```
```
True
```

> Syntax checking is achived when an expression matches with sequence building rule (sequence schema).
>
> This can be illustrated with a number phone.
>
> A phone number sequence is well-formed when the sequence :
>
> > - starts with 0 --> `^0`
> > - the zero is followed by a digit--> `^0([0-9])`
> > - a space follows with two digits --> `^0([0-9])([\s][0-9]{2})`
> > - this last sequence is repeated 4 times --> `^0([0-9])([\s][0-9]{2}){4}`.
>
> The expression built above allows to check if a french phone number is well-formed

In [30]:
```
sequence = '03 12 12 13 14'


phone_syntax_french = "^0([0-9])([\s][0-9]{2}){4}"
res_match = re.match(phone_syntax_french, sequence)
print('match' in str(type(res_match)).lower())
```
```
True
```

Find a pattern in a sequence: `re.findall`

> A pattern is a sequence of characters.
>
> In the example above, `111` and `33` may be considered as patterns.

In [32]:
```
sequence = "Hello 111 datasciences world 33! 11133"
expression = "([0-9]{3}|[0-9]{2})"
re.findall(expression, sequence)
```

Out[32]:
```
['111', '33', '111', '33']
```

> Even when attached, patterns are selectively found.

Unpacking

Step through two iterators at the same time

In [33]:
```
list_1 = ['a','b','c']# Iterator 1
list_2 = [1,2,3]# Iterator 2
for carac, number in zip(list_1, list_2) :
    print((carac, number))
```

```
('a', 1)
('b', 2)
('c', 3)
```

Using `*` character

In [180…
```
list_3 = [('a',1),('b',2),('c',3)]
list_char, list_number = zip(*list_3)
print(list_car)
print(list_number)
```

```
('a', 'b', 'c')
(1, 2, 3)
```

Unpacking function arguments

In [184…
```
def mult(a:int, b:int) :
    return a*b
```

In [185…
```
list_arg = [3,4]
print(mult(*list_arg))
```

```
12
```

Args et Kwargs

In [191…
```
def display_aguments(*args, **kwargs) :
    print('args = {}'.format(args))
    print("kwargs = {}".format(kwargs))
```

In [192…
```
display_aguments('a', 'b', 'c', value1 = 1, value2 = 2)
```

```
args = ('a', 'b', 'c')
kwargs = {'value1': 1, 'value2': 2}
```

> Any function defined with * `args` and ** `kwargs` accepts any number of anonymous or predefined arguments

Type `string`

```
In [205…
index = -1
index = "TOTO"[index+1:].find('O')
print(index)
index = "TOTO"[index+1:].find('O')
print(index)
index = "TOTO"[index+1:].find('O')
print(index)
```

```
1
1
1
```

```
In [206…
index = -1
index = "TOTO"[index+1:].find('O')
print(index)
```

```
1
```

```
In [207…
"TOTO"[index+1:]
```

```
Out[207]: 'TO'
```

> A string is an invariant

```
In [49]:
"TOTO"[1] = '8'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [49], in <cell line: 1>()
----> 1 "TOTO"[1] = '8'

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

Exercices

Exercice 1

> Write a function with annotations that detect wether or not followings python objects are iterator or not.
>
> Test function with following types : `list` , `dict` , `tuple`
>
> Create a generator and test it with your function.

Exercice 2

> Write functions that randomly replace a given character is an any string.
>
> - 1st function will use arguments : `replace_character_in_string(input_string, character)`

> - 2nd function will use `*args`
> - third one will use `**kwargs`

Exercice 3

> Define a generator that returns an even number for each iteration.

Exercice 4

> Create an expression that verifies that the followings URL are valid (or not).
>
> http://www.oliviertango.com or http://www.oliviertango.fr
>
> Define a function that implements the verification process.
>
> > - Function prints `TRUE` when valid, `FALSE` otherwise.
> > - Uses method `groups` from result of `re.search`
>
> Methodology :
>
> > - build your expression step by step and keep a track of each step.
> > - organize yourself, take time to define your plan to deliver this exercice

```
In [ ]:
```