Advanced python features

Computing Methods for Experimental Physics and Data Analysis

A. Manfreda

alberto.manfreda@pi.infn.it

INFN–Pisa

Compiled on October 28, 2019

Errors and Exceptions

- Error handling is one of the most important problem to solve when designing a program
- What should I do when I piece of code fails?
- What does fail mean?
  - Invalid input e.g. passing a path to a non existent file, or passing a string to a function for dividing numbers
  - Valid output not found, e.g searching the position of the letter 'd' in the string 'elephant'
  - Output cannot be find in a reasonable amount of time
  - Runtime resource failures: network connection down, disk space ended...
- Two phylosophies (historically):
  - Return some error flag (in different ways) to tell the user that something went wrong
  - Exceptions
- Example: a typical convention for programs is to return 0 from the main if the execution was successful and an error code (integer number) otherwise

Error flags

https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/error_flags.py

```
1  # The 'find()' method for strings in python uses an error flag
2  text = 'elephant'
3  print(text.find('p')) # upon success returns the position of the substring
4  print(text.find('d')) # returns -1 if the substring is not found
5
6  def safe_division(a, b):
7      if b == 0:
8          return 0 # Is that meaningful? What can we return?
9      else:
10         return a/b
11
12 # Why is this dangerous?
13 num_process = 0
14 num_cpu_available = 3
15 average_cpu_available = safe_division(num_cpu_available, num_process)
16 print(average_cpu_available) # Oops no cpu available... or not?
17
18 [Output]
19 3
20 -1
21 0
```

Problems of error flags

Error codes have their use (and are fine in some cases) but they suffer from a few issues:

- Choosing them is often arbitrary (and sometimes is difficult to make a sensible choice)
  - What if all the numbers can represent meaningful output of the function?
- Are cumbersome to use
  - Which error flag is used by a function? 0? -1? 99999999? $\rightarrow$ you have to go through the documentation for each!
  - If you have a deep hierarchy of functions you have to perform checks and pass the error up at every level!
- What if the caller of a function does not check the error flag?
  - The bug can propagate silently through its code!

We want something that:

- Is clearly separated from the returned output
- Cannot be silently ignored by the user
- Is easy to report to upper level without lots of lines of code

Enter exceptions

- An exception is an object that can be raised (in other languages also *thrown*) by a piece of code to signal that something went wrong
- When an exception is raised the normal flow of the code is interrupted
- The program automatically propagate the exception back in the function hierarchy until it found a place where the exception is catched and handled
- If the exception is never catched, not even in the main, the program crash with a specific error message
- Cathcing the exception is done with a *try - except* block

Exceptions

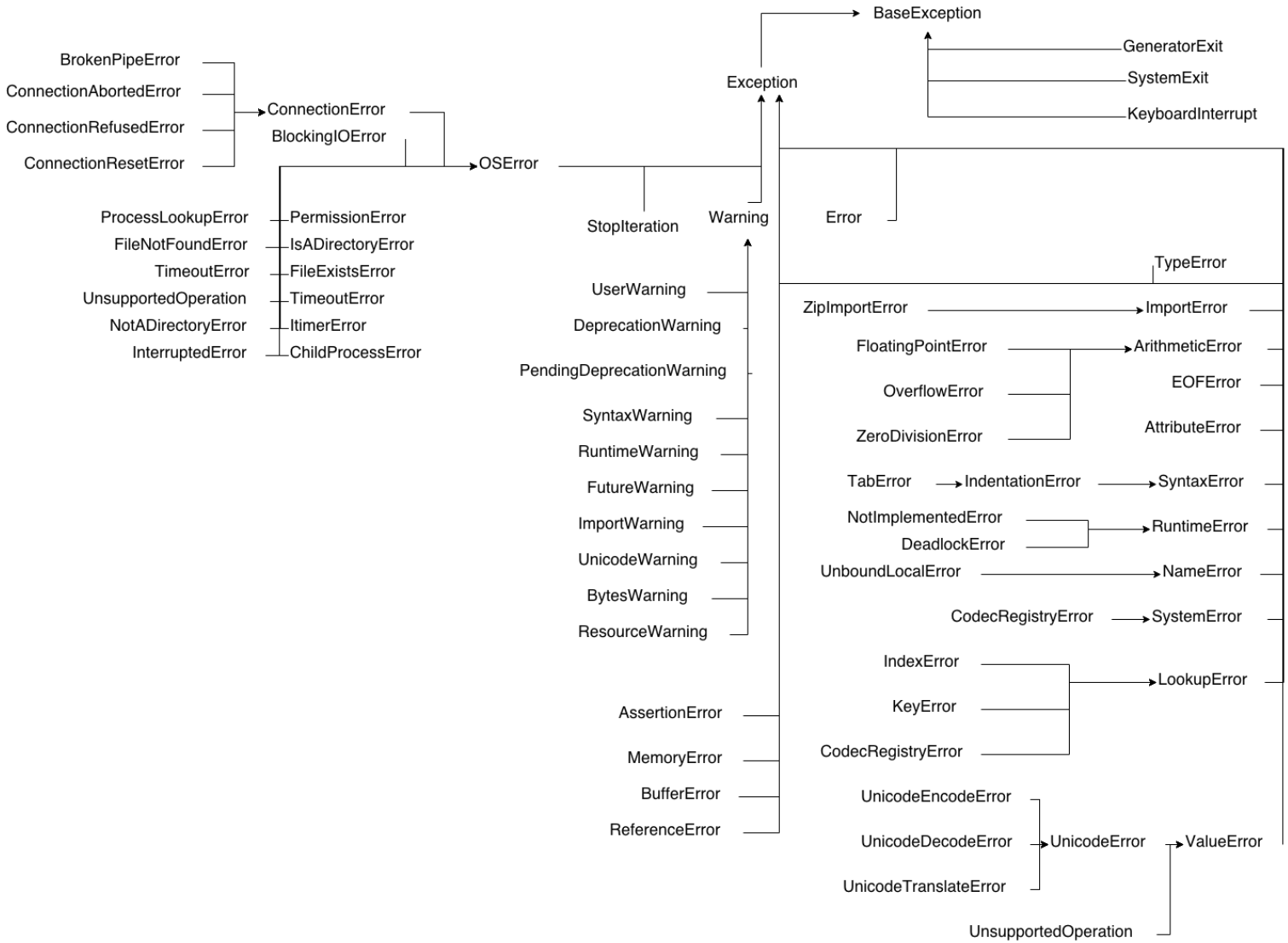
https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/exceptions.py

```
1  def throwing_function():
2      raise
3      print("This line is never executed!")
4
5  try:
6    throwing_function()
7    print("This line is never executed as well!")
8  except:
9    print("This line is executed only if an exception is raised in the try block")
10 finally: # optional!
11   print("This line is always executed")
12
13 [Output]
14 This line is executed only if an exception is raised in the try block
15 This line is always executed
```

The beauty of throwning stuff

- If that was all, exceptions would only be moderately useful
- The real bargain is that you can send back information together with the exception
- In fact you *are sending a full object*: the excetpion iteslf. Surprised?
- Inside the exception you can report all kind of data useful to reconstruct the exact error, which can be used by the caller for debug or to produce meaningful error messages
- You can also select which exceptions you catch, leaving the others propagate up
- Python provides a rich hierarchy of exception classes, which you can further customize (if you want) by deriving your own subclasses

The family tree of Python exceptions



Catching exceptions

https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/try_block.py

```
1  try:
2      with open ('i_do_not_exist.txt') as lab_data_file:
3          """ Do some process here...
4          ...
5          ...
6          """
7  except FileNotFoundError as e: # we assign a name to the the exception
8      print(e)
9
10
11 # We can be less specific by catching a parent exception
12 try:
13     with open ('i_do_not_exist.txt') as lab_data_file:
14         """ Do some process here...
15         ...
16         ...
17         """
18 except OSError as e: # OSError is a parent class of FileNotFoundError
19     print(e)
20
21 # catching Exception will catch almost everything!
22
23 [Output]
24 [Errno 2] No such file or directory: 'i_do_not_exist.txt'
25 [Errno 2] No such file or directory: 'i_do_not_exist.txt'
```

Throwing exceptions

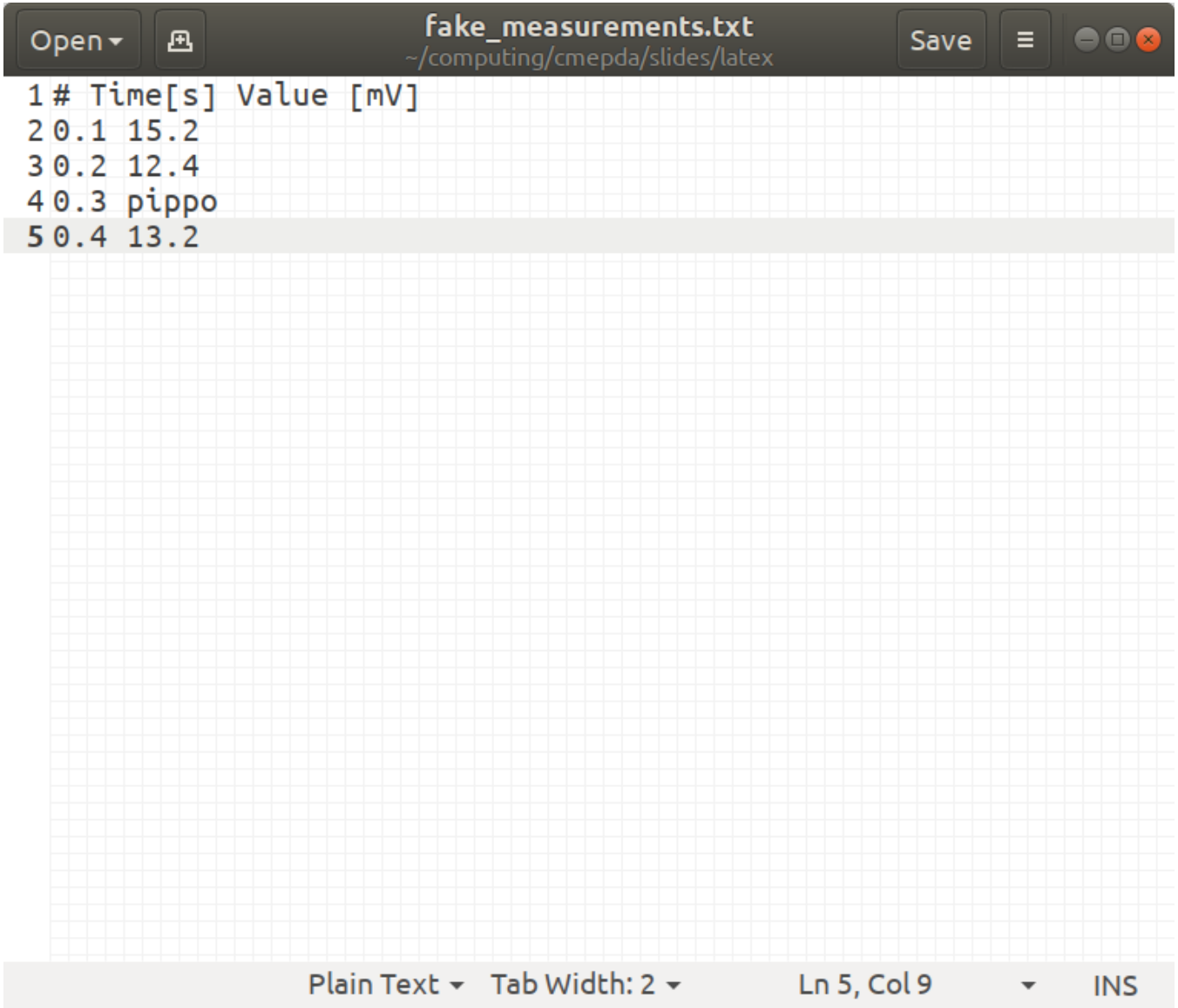
https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/throwing.py

```
1  def throwing_function():
2      # You can pass useful message to the exceptions you throw
3      raise RuntimeError('this is a useful debug text')
4
5  try:
6      throwing_function()
7  except RuntimeError as e:
8      # The message can be retrieved by printing the exception
9      print(e)
10
11 [Output]
12 this is a useful debug text
```

Where to catch exceptions?

- Differently from error flags, which needs to be checked as early as possible, you are not in a rush with exceptions
- Remeber: your goal is to provide the user a meaningful error message and useful debug information.
- You should catch an exception only when you have enough context to do that - which sometimes means waiting a few levels in the hierarchy!

When to catch?



When to catch?

https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/when_to_catch.py

```
1  def parse_line(line):
2      """ Parse a line of the file and return the values as float"""
3      values = line.strip('\n').split(' ')
4      # the following two lines may generate exceptions if they fails!
5      time = float(values[0])
6      tension = float(values[1])
7      return time, tension
8
9  with open('snippets/data/fake_measurements.txt') as lab_data_file:
10     for line in lab_data_file:
11         if not line.startswith('#'): # skip comments
12             time, tension = parse_line(line)
13             print(time, tension)
14
15 [Output]
16 0.1 15.2
17 0.2 12.4
18 Traceback (most recent call last):
19   File "snippets/when_to_catch.py", line 12, in <module>
20     time, tension = parse_line(line)
21   File "snippets/when_to_catch.py", line 6, in parse_line
22     tension = float(values[1])
23 ValueError: could not convert string to float: 'pippo'
```

Catch too early

https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/when_to_catch_1.py

```
1  def parse_line(line):
2      """ Parse a line of the file and return the values as float"""
3      values = line.strip('\n').split(' ')
4      try:
5          time = float(values[0])
6          tension = float(values[1])
7      except ValueError as e:
8          print(e) # This is not useful - which line of the file has the error?
9          return None # We can't really return something meaningful
10     return time, tension
11
12 with open('snippets/data/fake_measurements.txt') as lab_data_file:
13     for line in lab_data_file:
14         if not line.startswith('#'): # skip comments
15             time, tension = parse_line(line)
16             print(time, tension) # This line still crash badly!
17
18 [Output]
19 0.1 15.2
20 0.2 12.4
21 could not convert string to float: 'pippo'
22 Traceback (most recent call last):
23   File "snippets/when_to_catch_1.py", line 15, in <module>
24     time, tension = parse_line(line)
25 TypeError: 'NoneType' object is not iterable
```

Catch when needed

https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/when_to_catch_2.py

```
1  def parse_line(line):
2      """ Parse a line of the file and return the values as float"""
3      values = line.strip('\n').split(' ')
4      # the following two lines may generate exceptions if they fails!
5      time = float(values[0])
6      tension = float(values[1])
7      return time, tension
8
9  with open('snippets/data/fake_measurements.txt') as lab_data_file:
10     for line_number, line in enumerate(lab_data_file): # get the line number
11         if not line.startswith('#'): # skip comments
12             try:
13                 time, tension = parse_line(line)
14                 print(time, tension)
15             except ValueError as e:
16                 print('Line {} error: {}'.format(line_number, e))
17
18 [Output]
19 0.1 15.2
20 0.2 12.4
21 Line 3 error: could not convert string to float: 'pippo'
22 0.4 13.2
```

There is no check - only try

- In Python exceptions are the default methods for handling failures
- Many functions raise an exception when something goes wrong
- The common approach is: do not chech the input beforehand. Use it and be ready to catch excpetions if any.
- *Easier to ask for forgiveness than permission.*

Easier to ask for forgiveness

https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/dont_ask_permission.py

```
1  import os
2
3  file_path = 'i_do_not_exists.txt'
4
5  # Defensive version
6  if os.path.exists(file_path):
7      # What if the file is deleted between these two lines? (by another process)
8      # What if the file exists but you don't have permission to open it?
9      data_file = open(file_path)
10 else:
11     # Do something
12     print('Oops - file \'{}\' does not exist'.format(file_path))
13
14 # Pythonic way - you should prefer this one!
15 try:
16     data_file = open(file_path)
17 except OSError as e: # Cover more problems than FileNotFoundError
18     print('Oops - cannot read the file!\n{}'.format(e))
19
20 [Output]
21 Oops - file 'i_do_not_exists.txt' does not exist
22 Oops - cannot read the file!
23 [Errno 2] No such file or directory: 'i_do_not_exists.txt'
```

Iterators and iterables

- An *iterable* in Python is something that has a *__iter__* method, which returns an iterator
- An *iterator* is an object that implement a *__next__* method which is used to retrieve elements one at the time
- When there are no more elements to return, the iterator signals that with a specific exception: *StopIteration()*
- An iterator also implement an *__iter__* method that return. . . itself. So an iterator is also an iterable! (But the opposite is not true)

A 'for' loop unpacked

https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/show_iterator.py

```
1  my_list = [1., 2., 3.]
2
3  # For-loop syntax
4  for element in my_list:
5      print(element)
6
7  # This is equivalent (but much less readible and compact)
8  list_iterator = iter(my_list)
9  while True:
10     try:
11         print(next(list_iterator))
12     except StopIteration:
13         break
14
15 [Output]
16 1.0
17 2.0
18 3.0
19 1.0
20 2.0
21 3.0
```

A simple iterator

https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/simple_iterator.py

```
1  class SimpleIterator:
2      """ Class implementing a super naive iterator"""
3
4      def __init__(self, container):
5          self._container = container
6          self.index = 0
7
8      def __next__(self):
9          try:
10             # Note: here we are calling the __getitem__ method of self._container
11             item = self._container[self.index]
12         except IndexError:
13             raise StopIteration
14         self.index += 1
15         return item
16
17     def __iter__(self):
18         return self
19
20 class SimpleIterable:
21     """ A very basic iterable """
22
23     def __init__(self, *elements):
24         # We use a list to store elements internally.
25         # This provide us with the __getitem__ function
26         self._elements = list(elements)
27
28     def __iter__(self):
29         return SimpleIterator(self._elements)
```

A simple iterator

https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/test_simple_iterator.py

```
1  from simple_iterator import SimpleIterable
2
3  my_iterable = SimpleIterable(1., 2., 3., 'stella')
4  for element in my_iterable:
5      print(element)
6
7  [Output]
8  1.0
9  2.0
10 3.0
11 stella
```

A crazy iterator

https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/crazy_iterator.py

```
1  import random
2
3  class CrazyIterator:
4      """ Class implementing a crazy iterator"""
5
6      def __init__(self, container):
7          random.seed(1)
8          self._container = container
9
10     def __next__(self):
11         try:
12             # We get one possibility out of len(self._container) to exit
13             index = random.randint(0, len(self._container))
14             item = self._container[index]
15         except IndexError:
16             raise StopIteration
17         return item
18
19     def __iter__(self):
20         return self
21
22 class CrazyIterable:
23     """ Similar to a simple iterable, but with a twist... """
24
25     def __init__(self, *elements):
26         self._elements = list(elements)
27
28     def __iter__(self):
29         return CrazyIterator(self._elements)
```

A crazy iterator

https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/test_crazy_iterator.py

```
1  from crazy_iterator import CrazyIterable
2
3  my_iterable = CrazyIterable('A', 'B', 'C', 'D', 'E')
4  for element in my_iterable:
5      print(element)
6
7  [Output]
8  B
9  E
10 A
11 C
12 A
13 D
14 D
15 D
```

Python tools for iterables

- Python provides a number of functions that consume an iterable and return a single value:
  - *sum*: Sum all the elements
  - *all*: Return true if a given condition is true for all the elements
  - *any*: Return true if a given condition is true for at lest one element
  - *max*: Return the max
  - *min*: Return the minimum
- In addition, in the *functools* library there is the generic *reduce* function that works as follow:
  - Apply a given function to the first two elements
  - Apply the function to the result of the first evaluation and the third element
  - Continue like that until the iterable is over, than return the result

Generators

- We have seen that iterators are useful to iterate over container
- However that assumes a containers exists → memory usage
- Generators allow you to loop over sequences of items even when they don't exist before - they itrms are just created lazily the moment they are required
- For example you can write a generator to loops over the Fibonacci succession. You can't create the sequence earlier, since it is not finite!
- Generators are created through either generator expressions or generator functions
- In real life most of the time you will simply use pre-made functions that return a generator, like *range()* (in Python 3)
- Generator can be used to iterate in for loops, just like iterators

Generators first look

https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/generators.py

```
1  """ range() is a function that returns a generator in Python 3. The list of
2  numbers never exists entirely, they are created ine at a time.
3  Note: In Python 2 range() does create the full list at the beginning.
4  There used to be a xrange() function for lazy generation, which is now
5  deprecated in Python 3. """
6  for i in range(4):  # generators act like iterators in for loop
7      print(i)
8
9  data = [12, -1, 5]
10 square_data_generator = (x**2 for x in data) # generator expression!
11 for square_datum in square_data_generator: # again,
12     print(square_datum)
13
14 [Output]
15 0
16 1
17 2
18 3
19 144
20 1
21 25
```

Generator functions

- A generator function is a function that contains the keyword yield at least once in his body
- When you call a generator function the code is not executed - instead a generator object is created and returned (even if you don't have a return statement)
- Each call to *next()* on the returned generator will make the function code run until it finds a yield statement
- Then the execution is paused and the value of the expression on the right of yiel is returned (yielded) to the caller
- A further call of next will resume the execution from where it was suspended until the next yield and so on
- Eventually, when the function body ends, *StopIteration* is raised
- Usually generators functions contain a loop - but it's not mandatory!

Generator functions

https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/generator_functions.py

```
1  def generator_function_simple():
2      print('First call')
3      yield 1+1
4      print('Second call')
5      yield ['Darth', 'Vader']
6      print('I am about to rise an exception...')
7
8  gen = generator_function_simple() # A generator function returns a generator
9  print(next(gen)) # We stop at the first yield and get the value
10 print(next(gen)) # Second yield
11 next(gen) # The third next() will throw StopIteration
12
13 [Output]
14 First call
15 2
16 Second call
17 ['Darth', 'Vader']
18 I am about to rise an exception...
19 Traceback (most recent call last):
20   File "snippets/generator_functions.py", line 11, in <module>
21     next(gen) # The third next() will throw StopIteration
22 StopIteration
```

Infinite sequence generators

https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/fibonacci.py

```
1  # Generator function that provides infinte fibonacci numbers
2  def fibonacci():
3      a, b = 0, 1
4      while True:
5          yield a
6          a, b = b, a + b
7
8  # We need to impose a stop condition externally to use it
9  max_n = 7
10 fib_numbers = []
11 for i, fib in enumerate(fibonacci()):
12     if i >= max_n:
13         break
14     else:
15         fib_numbers.append(fib)
16 print(fib_numbers)
17
18 # Another way to do that  is using 'islice' from itertools
19 import itertools
20 # Generator expression - note the +1 in islice
21 fib_gen = (fib for fib in itertools.islice(fibonacci(), max_n))
22 print(list(fib_gen))
23
24 [Output]
25 [0, 1, 1, 2, 3, 5, 8]
26 [0, 1, 1, 2, 3, 5, 8]
```

Python generator functions

- Python provides a number of built-in functions that return a generator from an iterable, such as:
  - *enumerate*: Automatic counting of iterations
  - *map*: Apply a function to the elements
  - *filter*: Return only the elements passing a given condition
  - *zip*: Return pairs of elements (requires two sequences)
  - *reversed*: Loop in the reversed order
- Countless others can be found in the *itertools* library
  - *islice*: Slice the loop with start, stop and step
  - *takewhile*: Stop looping when a condition becomes false
  - *accumulate*: Get the result of applying the function iteratively to pair of elements
  - *chain*: Loop through many sequences one after another
  - *cycle*: Loop over the sequence repeatedly, indefinitely
  - *permutations*: Get all the permutations of a given length
  - And so on...
- Take a look at the documentation of each function to see how to properly call it!

Anonymous (lambda) functions

- Anonymous functions, or lambda functions are a construct typical of functional programming
- `https://en.wikipedia.org/wiki/Lambda_calculus`
- `https://en.wikipedia.org/wiki/Functional_programming`
- In Python a lambda function is essentially a special sintax for creating a function on the fly, without giving it a name
- They are limited to a single expression, which is returned to the user
- Many of the typical uses for lambdas are already covered in python by generator expressions and comprehension, so this is more like a niche feature of the language

Lambda functions

https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/lambda.py

```
1  # Here we create a lambda function and assign a name to it (ironically)
2  multiply = lambda x, y: x * y
3  # Use it
4  print(multiply(5, -1))
5
6  # Typical use is inside generator functions
7  numbers = range(10)
8  squares = list(map(lambda n: n**2, numbers))
9  print(squares)
10
11 # However, remeber that you can do the same with list comprehension
12 squares = [n**2 for n in numbers]
13 print(squares)
14
15 [Output]
16 -5
17 [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
18 [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```