Advanced python features

Computing Methods for Experimental Physics and Data Analysis

A. Manfreda

alberto.manfreda@pi.infn.it

INFN–Pisa

Compiled on October 13, 2021

▷ 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

```python
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   # Why is this dangerous?
7   def cut_before(input_string, substring):
8       """ Cut a string from the beginning up to the position before that of
9       the given substring, then return it """
10      pos = input_string.find(substring)
11      return input_string[:(pos)]
12
13  # If the substring exists in the string everything works fine
14  print(cut_before('We all live in a Yellow Submarine', 'Yellow'))
15  # What will be the output here?
16  print(cut_before('We all live in a Yellow Submarine', 'Red'))
17
18  [Output]
19  3
20  -1
21  We all live in a
22  We all live in a Yellow Submarin
```

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? → 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

```python
1   # index() is the same as find(), but rise an exception in case of failure
2   def cut_before(input_string, substring):
3       """ Cut a string from the beginning up to the position before that of
4       the given substring, then return it """
5       pos = input_string.index(substring)
6       return input_string[:(pos)]
7
8   # If the substring exists in the string everything works fine
9   print(cut_before('We all live in a Yellow Submarine', 'Yellow'))
10  # No silent bug here!
11  print(cut_before('We all live in a Yellow Submarine', 'Red'))
12
13  [Output]
14  We all live in a
15  Traceback (most recent call last):
16    File "snippets/exceptions_vs_err_flags.py", line 11, in <module>
17      print(cut_before('We all live in a Yellow Submarine', 'Red'))
18    File "snippets/exceptions_vs_err_flags.py", line 5, in cut_before
19      pos = input_string.index(substring)
20  ValueError: substring not found
```

'Exception, Your Honor!'

▷ 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

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

```
1   def cut_before(input_string, substring):
2       try:
3           result = input_string[:(input_string.index(substring))]
4           print('This line is not executed if an exception is raised in the try block')
5           return result
6       # Catch the correct exception type with 'except'
7       except ValueError:
8           print('This line is executed only if a ValueError is raised in the try block')
9
10  cut_before('We all live in a Yellow Submarine', 'Yellow')
11  cut_before('We all live in a Yellow Submarine', 'Red')
12
13  [Output]
14  This line is not executed if an exception is raised in the try block
15  This line is executed only if a ValueError is raised in the try block
```

▷ There are two more optional statements in a try-block:

  ▷ *else*: executed only if no exception is raised in the try block

  ▷ *finally*: executed no matter what

▷ *finally* is executed even if there is a return statement in the try block

▷ can be used to release important resources (e.g. closing a file, or a connection)

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

```python
 1   def cut_before(input_string, substring):
 2       try:
 3           result = input_string[:(input_string.index(substring))]
 4           print('This line is not executed if an exception is raised in the try block')
 5       except ValueError:
 6           print('This line is executed only if a ValueError is raised in the try block')
 7       else: # optional!
 8           print('This line is executed only if no exception is raised in the try block')
 9           return result
10       finally: # optional!
11           print('This line is always executed')
12
13   cut_before('We all live in a Yellow Submarine', 'Yellow')
14   cut_before('We all live in a Yellow Submarine', 'Red')
15
16   [Output]
17   This line is not executed if an exception is raised in the try block
18   This line is executed only if no exception is raised in the try block
19   This line is always executed
20   This line is executed only if a ValueError is raised in the try block
21   This line is always executed
```
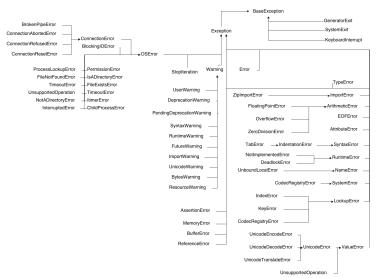
The beauty of exceptions

▷ 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

```
   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           pass
 6
 7   except FileNotFoundError as e: # we assign a name to the the exception
 8       print(e)
 9
10   # We can be less specific by catching a parent exception
11   except OSError as e: # OSError is a parent class of FileNotFoundError
12       print(e)
13
14   # catching Exception will catch almost everything!
15   except Exception as e:
16       print(e)
17
18   [Output]
19   [Errno 2] No such file or directory: 'i_do_not_exist.txt'
```

▷ Warning: catching *Exception*, will also catch *SyntaxError* and *NameError*

▷ This mean that the code will 'run' even if there is a typo in it!

▷ Bottom line: you should never catch generically for *Exception*, always be more specific

▷ Even worse, you should never catch for *BaseException* as that would even prevent the user for from aborting the execution with a *KeyboardInterrupt* (e.g. Ctrl-C)

▷ Unless that is what you need, of course

▷ 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 exceptions if any.

▷ *Easier to ask for forgiveness than permission.*

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

```python
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'
```

▷ A stack traceback, or simply traceback, is a list of the function calls at a specific point in the code

▷ An exception always stores the traceback at the failure point, since it contains crucial debug information

▷ In Python a traceback is read from the bottom up:

  ▷ The last line of the traceback is the error message line, which contains the name of the exception that was raised

  ▷ Further up are the various function calls from most recent to least recent

▷ When you catch an exception, the content of the traceback can be displayed using either the *logging* module or the *traceback* module

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

```python
1   """ Showing the tracebak with logging"""
2   import logging
3   logger = logging.Logger('demo')
4
5   try:
6       with open ('i_do_not_exist.txt') as myfile:
7           pass
8   except FileNotFoundError as e:
9       logger.exception(e)
10
11  [Output]
12  [Errno 2] No such file or directory: 'i_do_not_exist.txt'
13  Traceback (most recent call last):
14    File "snippets/traceback_logging.py", line 6, in <module>
15      with open ('i_do_not_exist.txt') as myfile:
16  FileNotFoundError: [Errno 2] No such file or directory: 'i_do_not_exist.txt'
```

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

```
1   """ Printing the traceback with the 'traceback' module """
2   import traceback
3   import sys
4
5   try:
6       with open ('i_do_not_exist.txt') as myfile:
7           pass
8   except FileNotFoundError as e:
9       # The tracebak is stored as an attribute of the exception
10      # By default print_tb() output is redirected to the stderr, but you can
11      # change that by setting the 'file' argument
12      traceback.print_tb(e.__traceback__, file=sys.stdout)
13
14  [Output]
15    File "snippets/traceback_tb.py", line 6, in <module>
16      with open ('i_do_not_exist.txt') as myfile:
```

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

```python
1   """ Printing the exception with the 'traceback' module """
2   import traceback
3   import os
4
5   try:
6       with open ('i_do_not_exist.txt') as myfile:
7           pass
8   except FileNotFoundError as e:
9       # Log exception to file
10      with open ('traceback.log', 'w') as logfile:
11          # This will print the full exception
12          traceback.print_exception(e.__class__, e, e.__traceback__, file=logfile)
13          # Since Python 3.10 this was simplified into:
14          # traceback.print_exception(e)
15      os.system('cat traceback.log')
16
17  [Output]
18  Traceback (most recent call last):
19    File "snippets/traceback_exception.py", line 6, in <module>
20      with open ('i_do_not_exist.txt') as myfile:
21  FileNotFoundError: [Errno 2] No such file or directory: 'i_do_not_exist.txt'
```

▷ Up to now we have been dealing with exceptions generated by Python functions

▷ What about raising exceptions ourselves?

Raising exceptions

```python
def raising_function():
    # You can pass an useful message to the exceptions you raise
    raise RuntimeError('this is a useful debug message')

try:
    raising_function()
except RuntimeError as e:
    # The message can be retrieved by printing the exception
    print(e)

[Output]
this is a useful debug message
```

▷ Beside the built-in exceptions provided by Python, you can add your own custom exceptions by inheriting from the *Exception* class

▷ This serves two purposes:
  ▷ Make the exception handling code more specific, and hence more readible
  ▷ Allows you to pass additional data with your exception - in the form of attributes of the class - which can be used for debug or any other purpose

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

```
 1   class SimpleCustomError(Exception):
 2       pass # Yeah that's it
 3
 4   raise SimpleCustomError('simple error')
 5
 6   [Output]
 7   Traceback (most recent call last):
 8     File "snippets/custom_exceptions.py", line 4, in <module>
 9       raise SimpleCustomError('simple error')
10   __main__.SimpleCustomError: simple error
```

https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/custom_exceptions_2.py

```
 1   class ValueTooLargeError(ValueError):
 2       def __init__(self, value):
 3           self.value = value
 4           super().__init__('{}: {} is too large'.format(self.__class__.__name__,
 5                                                          self.value))
 6
 7   value = 100
 8   try:
 9     if value > 10:
10         raise ValueTooLargeError(value)
11   except ValueError as e:
12       print(e)
13
14   [Output]
15   ValueTooLargeError: 100 is too large
```
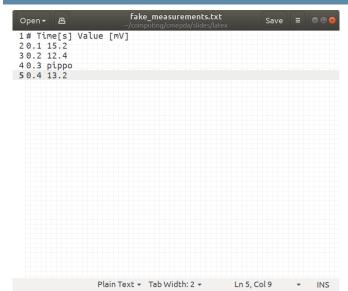
▷ Differently from error flags, which need to be checked as early as possible, you are not in a rush with exceptions

▷ 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?

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

```python
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 fail!
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'
```

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

```python
def parse_line(line):
    """ Parse a line of the file and return the values as float"""
    values = line.strip('\n').split(' ')
    try:
        time = float(values[0])
        tension = float(values[1])
    except ValueError as e:
        print(e) # This is not useful - which line of the file has the error?
        return None # We can't really return something meaningful
    return time, tension

with open('snippets/data/fake_measurements.txt') as lab_data_file:
    for line in lab_data_file:
        if not line.startswith('#'): # skip comments
            time, tension = parse_line(line)
            print(time, tension) # This line still crash badly!

[Output]
0.1 15.2
0.2 12.4
could not convert string to float: 'pippo'
Traceback (most recent call last):
  File "snippets/when_to_catch_1.py", line 15, in <module>
    time, tension = parse_line(line)
TypeError: 'NoneType' object is not iterable
```

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

```python
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       time = float(values[0])
5       tension = float(values[1])
6       return time, tension
7
8   with open('snippets/data/fake_measurements.txt') as lab_data_file:
9       for line_number, line in enumerate(lab_data_file): # get the line number
10          if not line.startswith('#'): # skip comments
11              try:
12                  time, tension = parse_line(line)
13                  print(time, tension)
14              except ValueError as e:
15                  print('Line {} error: {}'.format(line_number, e))
16
17  [Output]
18  0.1 15.2
19  0.2 12.4
20  Line 3 error: could not convert string to float: 'pippo'
21  0.4 13.2
```

Iterators

▷ 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 technically an iterable[1]! (But the opposite is not true)

---

[1]Only 'technically' because an iterator has no data of its own, so you always need a 'real' iterable to actually iterate

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 readable 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
```

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

```python
 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)
```

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
```

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

```python
import random

class CrazyIterator:
    """ Class implementing a crazy iterator"""

    def __init__(self, container):
        random.seed(1)
        self._container = container

    def __next__(self):
        try:
            # We get one possibility out of len(self._container) to exit
            index = random.randint(0, len(self._container))
            item = self._container[index]
        except IndexError:
            raise StopIteration
        return item

    def __iter__(self):
        return self

class CrazyIterable:
    """ Similar to a simple iterable, but with a twist... """

    def __init__(self, *elements):
        self._elements = list(elements)

    def __iter__(self):
        return CrazyIterator(self._elements)
```

```
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 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
  ▷ *functools.reduce*: Apply a function recursively to pairs of elements

Generators

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 - the items 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

```python
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, works like an iterator
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 *yield* 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!

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

```python
1   def generator_function_simple():
2       print('First call')
3       yield 1
4       print('Second call')
5       yield 2
6       print('I am about to rise a StopIteration 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  1
16  Second call
17  2
18  I am about to rise a StopIteration 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
```

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 of doing that is using 'islice' from itertools
19   import itertools
20   # Generator expression
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 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 results 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
  ▷ *product*: Compute the cartesian product of iterables
  ▷ *groupby*: Group by value of some key (function)
  ▷ And so on. . .

▷ Take a look at the documentation of each function to see how to
properly call it!

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

```python
1   from itertools import accumulate, product, chain, groupby, permutations, combinations
2   import operator
3
4   l1 = [1, 2, 3, 4]
5   print(list(accumulate(l1)))
6   print(list(accumulate(l1, func=operator.mul)))
7   print(list(combinations(l1, 3)))
8
9   l2 = [5, 6]
10  print(list(permutations(l2, 2)))
11  print(list(product(l1, l2)))
12
13  def is_even(n):
14      return n % 2 == 0
15
16  l3 = list(chain(l1, l2))
17  # groupby expect the list to be sorted by the grouping function
18  l3.sort(key=is_even)
19  for k, g in groupby(l3, key=is_even):
20      print(k, list(g))
21
22  [Output]
23  [1, 3, 6, 10]
24  [1, 2, 6, 24]
25  [(1, 2, 3), (1, 2, 4), (1, 3, 4), (2, 3, 4)]
26  [(5, 6), (6, 5)]
27  [(1, 5), (1, 6), (2, 5), (2, 6), (3, 5), (3, 6), (4, 5), (4, 6)]
28  False [1, 3, 5]
29  True [2, 4, 6]
```

Lambda functions

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

```
                        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]
```

```python
from itertools import dropwhile


class LabFileIterator:
    def __init__(self, file_obj):
        self._lines = dropwhile(lambda line: line.startswith('#'), file_obj)

    def __next__(self):
        line = next(self._lines)
        values = line.strip('\n').split(' ')
        time = float(values[0])
        tension = float(values[1])
        return time, tension

    def __iter__(self):
        return self


with open('snippets/data/fake_measurements.txt') as lab_data_file:
    try:
        for line_number, (time, tension) in enumerate(LabFileIterator(lab_data_file)):
            print(line_number, time, tension)
    except ValueError as e:
        # Here we get the wrong line number! Why?
        print('Line {} error: {}'.format(line_number, e))

[Output]
0 0.1 15.2
1 0.2 12.4
Line 1 error: could not convert string to float: 'pippo'
```

https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/file_iterator_2.py

```python
 1   from itertools import dropwhile
 2
 3   class LabFile:
 4       def __init__(self, file_obj):
 5           self._file = file_obj
 6
 7       def __iter__(self):
 8           # Enumerate is now inside dropwhile, so all lines are counted
 9           for i, line in dropwhile(lambda x : x[1].startswith('#'),
10                                    enumerate(self._file)):
11               values = line.strip('\n').split(' ')
12               try:
13                   time = float(values[0])
14                   tension = float(values[1])
15               except ValueError as e:
16                   print('Line {} error: {}'.format(i, e))
17                   continue
18               yield time, tension
19
20   with open('snippets/data/fake_measurements.txt') as lab_data_file:
21       for time, tension in LabFile(lab_data_file):
22           print(time, tension)
23
24   [Output]
25   0.1 15.2
26   0.2 12.4
27   Line 3 error: could not convert string to float: 'pippo'
28   0.4 13.2
```

https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/file_iterator_2.py

```python
from itertools import dropwhile

class LabFile:
    def __init__(self, file_obj):
        self._file = file_obj

    def __iter__(self):
        # Enumerate is now inside dropwhile, so all lines are counted
        for i, line in dropwhile(lambda x : x[1].startswith('#'),
                                 enumerate(self._file)):
            values = line.strip('\n').split(' ')
            try:
                time = float(values[0])
                tension = float(values[1])
            except ValueError as e:
                print('Line {} error: {}'.format(i, e))
                continue
            yield time, tension

with open('snippets/data/fake_measurements.txt') as lab_data_file:
    for time, tension in LabFile(lab_data_file):
        print(time, tension)

[Output]
0.1 15.2
0.2 12.4
Line 3 error: could not convert string to float: 'pippo'
0.4 13.2
```