Advanced python features

Computing Methods for Experimental Physics and Data Analysis

A. Manfreda

alberto.manfreda@pi.infn.it

INFN–Pisa

Compiled on October 14, 2020

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

```
# The ’find()’ method for strings in python uses an error flag
text = ’elephant’
print(text.find(’p’)) # upon success returns the position of the substring
print(text.find(’d’)) # returns -1 if the substring is not found

# Why is this dangerous?
def cut_two_before(input_string, substring):
    """ Cut a string up to two positions before that of the given substring and
    return it """
    pos = input_string.find(substring)
    return input_string[:(pos-1)]

# If the substring exists in the string everything works fine
print(cut_two_before(’We all live in a Yellow Submarine’, ’Yellow’))
# What will be the output here?
print(cut_two_before(’We all live in a Yellow Submarine’, ’Red’))

[Output]
3
-1
We all live in a
We all live in a Yellow Submari
```

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? → 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

A different way

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

```python
# index() is the same as find(), but rise an exception in case of failure
def cut_two_before(input_string, substring):
    """ Cut a string up to two positions before that of the given substring and
    return it """
    pos = input_string.index(substring)
    return input_string[:(pos-1)]

# If the substring exists in the string everything works fine
print(cut_two_before('We all live in a Yellow Submarine', 'Yellow'))
# No silent bug here!
print(cut_two_before('We all live in a Yellow Submarine', 'Red'))

[Output]
We all live in a
Traceback (most recent call last):
  File "snippets/exceptions_vs_err_flags.py", line 11, in <module>
    print(cut_two_before('We all live in a Yellow Submarine', 'Red'))
  File "snippets/exceptions_vs_err_flags.py", line 5, in cut_two_before
    pos = input_string.index(substring)
ValueError: substring not found
```

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

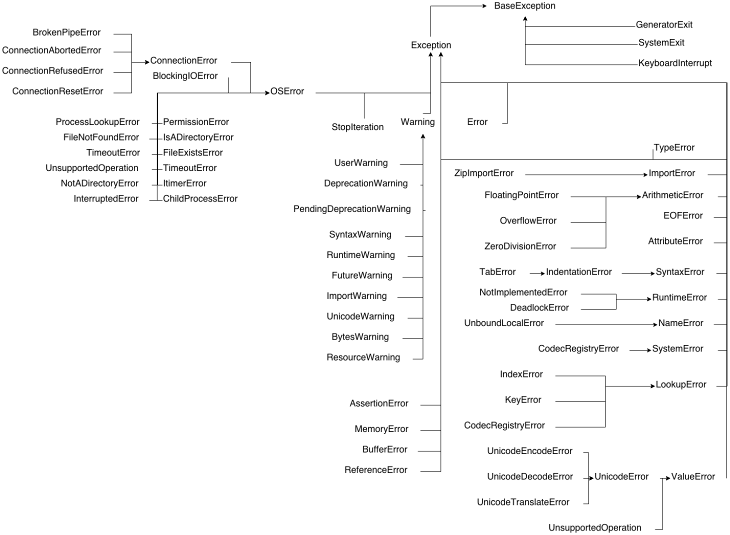
https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/exceptions.py

```python
def throwing_function():
    raise
    print("This line is never executed!")

try:
  throwing_function()
  print("This line is never executed as well!")
except:
  print("This line is executed only if an exception is raised in the try block")
else: # optional!
  print("This line is executed only if no exception is raised in the try block")
finally: # optional!
  print("This line is always executed")

[Output]
This line is executed only if an exception is raised in the try block
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



BaseException
GeneratorExit
SystemExit
KeyboardInterrupt
Exception
BrokenPipeError
ConnectionAbortedError
ConnectionRefusedError
ConnectionResetError
ConnectionError
BlockingIOError
OSError
ProcessLookupError
FileNotFoundError
TimeoutError
UnsupportedOperation
NotADirectoryError
InterruptedError
PermissionError
IsADirectoryError
FileExistsError
TimeoutError
ItimerError
ChildProcessError
StopIteration
Warning
Error
TypeError
UserWarning
DeprecationWarning
PendingDeprecationWarning
SyntaxWarning
RuntimeWarning
FutureWarning
ImportWarning
UnicodeWarning
BytesWarning
ResourceWarning
ZipImportError
ImportError
FloatingPointError
OverflowError
ZeroDivisionError
ArithmeticError
EOFError
AttributeError
TabError
IndentationError
SyntaxError
NotImplementedError
DeadlockError
RuntimeError
UnboundLocalError
NameError
CodecRegistryError
SystemError
IndexError
KeyError
CodecRegistryError
LookupError
AssertionError
MemoryError
BufferError
ReferenceError
UnicodeEncodeError
UnicodeDecodeError
UnicodeTranslateError
UnicodeError
ValueError
UnsupportedOperation

Catching exceptions

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

```
try:
    with open (’i_do_not_exist.txt’) as lab_data_file:
        """ Do some process here...
        """
        pass

except FileNotFoundError as e: # we assign a name to the the exception
    print(e)

# We can be less specific by catching a parent exception
except OSError as e: # OSError is a parent class of FileNotFoundError
    print(e)

# catching Exception will catch almost everything!
except Exception as e:
    print(e)

[Output]
[Errno 2] No such file or directory: ’i_do_not_exist.txt’
```

Raising exceptions

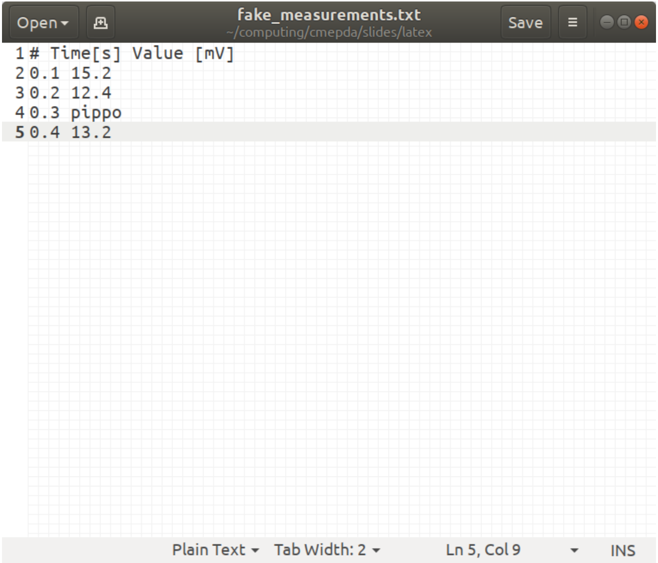
https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/raising.py

```python
def raising_function():
    # You can pass 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
```

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?



```
# Time[s] Value [mV]
0.1 15.2
0.2 12.4
0.3 pippo
0.4 13.2
```

When to catch?

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

```
def parse_line(line):
    """ Parse a line of the file and return the values as float"""
    values = line.strip(’\n’).split(’ ’)
    # the following two lines may generate exceptions if they fails!
    time = float(values[0])
    tension = float(values[1])
    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)

[Output]
0.1 15.2
0.2 12.4
Traceback (most recent call last):
  File "snippets/when_to_catch.py", line 12, in <module>
    time, tension = parse_line(line)
  File "snippets/when_to_catch.py", line 6, in parse_line
    tension = float(values[1])
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

```
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
```

Catch when needed

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

```
def parse_line(line):
    """ Parse a line of the file and return the values as float"""
    values = line.strip(’\n’).split(’ ’)
    # the following two lines may generate exceptions if they fails!
    time = float(values[0])
    tension = float(values[1])
    return time, tension

with open(’snippets/data/fake_measurements.txt’) as lab_data_file:
    for line_number, line in enumerate(lab_data_file): # get the line number
        if not line.startswith(’#’): # skip comments
            try:
                time, tension = parse_line(line)
                print(time, tension)
            except ValueError as e:
                print(’Line {} error: {}’.format(line_number, e))

[Output]
0.1 15.2
0.2 12.4
Line 3 error: could not convert string to float: ’pippo’
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

```python
import os

file_path = ’i_do_not_exists.txt’

# Defensive version
if os.path.exists(file_path):
    # What if the file is deleted between these two lines? (by another process)
    # What if the file exists but you don’t have permission to open it?
    data_file = open(file_path)
else:
    # Do something
    print(’Oops - file \’{}\’ does not exist’.format(file_path))

# Pythonic way - you should prefer this one!
try:
    data_file = open(file_path)
except OSError as e: # Cover more problems than FileNotFoundError
    print(’Oops - cannot read the file!\n{}’.format(e))

[Output]
Oops - file ’i_do_not_exists.txt’ does not exist
Oops - cannot read the file!
[Errno 2] No such file or directory: ’i_do_not_exists.txt’
```

Decorators

Functions inside functions

- Function in python are first class object
- The name is a bit misleading, but what actually means is that functions can be passed as argument to other functions and returned as result from other functions
- This shouldn't surprise you much: functions are objects of a '*function*' class, so they behave like any other vairable in Python
- Another thing you can (and sometimes want to) do is *defining* a function inside another.
- Let's see how it works

Functions inside functions

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

```
def outer():
    def inner(): # Defining the inner function inside the outer function
        print(’Inner function’)
        return # End of the inner function
    return inner # Inner is the output of outer

my_func = outer() # my_func is now refercing ’inner’
print(my_func.__name__)
my_func() # Calling my_func is equal to calling ’inner’

def outer2():
    some_string = ’Hello!’
    def inner():
        # We have access to the variables in the outer function!
        print(some_string)
    return inner

my_other_func = outer2()
my_other_func()

[Output]
inner
Inner function
Hello!
```

Colsures and free variables

- When a function is created inside another function it has access to the local variables of the outer function, even after its scope ended
- This is techincally possible because those varibales are kept in a special space of memory, the closure of the inner function
- Such variables are called free variables
- Note: if you *assign* to a free variable in the inner function, by default a new, local variable is created instead!
- To avoid this you have to explictly declare that you want to access the variable in the closure using the nonlocal keyword
- Remember: *'Explicit is better then implicit'*

Free variables - a mistake to avoid

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

```
def running_average():
    total_count = 0
    num_elements = 0
    def accumulator(value):
        total_count += value # Doesn’t work! total_count is reassigned!
        num_elements += 1 # Doesn’t work! total_count is reassigned!
        return total_count/num_elements
    return accumulator

run_avg = running_average()
print(run_avg(1.))
print(run_avg(5.))
print(run_avg(2.5))

[Output]
Traceback (most recent call last):
  File "snippets/closure_wrong.py", line 11, in <module>
    print(run_avg(1.))
  File "snippets/closure_wrong.py", line 5, in accumulator
    total_count += value # Doesn’t work! total_count is reassigned!
UnboundLocalError: local variable ’total_count’ referenced before assignment
```

Free variables - the correct way

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

```
def running_average():
    total_count = 0
    num_elements = 0
    def accumulator(value):
        # We declare the relevant variables as nonlocal
        nonlocal total_count, num_elements
        # Now we can assign to them - the variables in the closure will be
        # modified, as we want!
        total_count += value
        num_elements += 1
        return total_count/num_elements
    return accumulator

run_avg = running_average()
print(run_avg(1.))
print(run_avg(5.))
print(run_avg(2.5))

[Output]
1.0
3.0
2.8333333333333335
```

Wrapping functions

- The typical use of defining a function inside a function is to create a wrapper
- A wrapper is a function that calls another one adding a layer of functionalities in between - for example it may do some pre-process of the input, or change the output in some way, or measure the execution time or whatever we want
- The techinque for creating a wrapper fucntion in Python is:
  - Pass the function that we want to wrap as argument of the outer function
  - Inside the outer function we define an inner function, which is the actual wrapper
  - The wrapper calls the wrapped function and adds its functionalities, before and/or after the call. It may return the same output or a manipulated one.
  - Then from the outer fucntion we return the wrapper

Wrapper

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

```python
def some_function(a, b):
    print('Executing {} x {}'.format(a, b))
    return a * b

def add_n_wrapper(func, n): # We take the wrapped function as argument
    """ This wrapper adds n to the result of the wrapped function"""

    def wrapper(*args, **kwargs):
       """We passs the arguments as *arg, **kwargs, because this is the most
       general form in Python: we can collect any comination of arguments like
       that. Note that we have access to both 'func' and 'n', as they are stored
       in the closure of 'wrapper'"""
       result = func(*args, **kwargs) # Pass the arguments to the wrapped fucntion
       print('Adding {}'.format(n))
       return result + n # Return a modified result in this case

    return wrapper # From add_n_wrapper we return the wrapper

function_plus_five = add_n_wrapper(some_function, 5)
print('Result = {}'.format(function_plus_five(2, 3)))

[Output]
Executing 2 x 3
Adding 5
Result = 11
```

Decorators

- Often, when you wrap a function, you don't want to change it's name, so you reassign the wrapped funtion to its old name
- In fact, this techinque is so common that python introduced a special syntax for it: decorators
- A decorated function has simply the name of the wrapper added with a '@' on top of its declaration

Decorators

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

```python
def print_function_info(func):
    def wrapper(*args, **kwargs):
        print('Calling function \'{}\''.format(func.__name__))
        print('Positional arguments = {}'.format(args))
        print('Keyword arguments = {}'.format(kwargs))
        return func(*args, **kwargs)
    return wrapper

@print_function_info
def some_function(a, b, c=0):
    return a * b + c

# This is equivalent to: some_function = print_function_info(some_function)

print(some_function(1, 2, c=7))
 # Inspecting the function reveals that we are calling the wrapper
print('The name of the function is \'{}\''.format(some_function.__name__))

[Output]
Calling function 'some_function'
Positional arguments = (1, 2)
Keyword arguments = 'c': 7
9
The name of the function is 'wrapper'
```

A decorator to measure execution time

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

```
import time
from functools import wraps

def clocked(func):
    """ We use functools.wraps to keep the original function name and docstring"""
    @wraps(func)
    def wrapper(*args, **kwargs):
        tstart = time.clock()
        result = func(*args, **kwargs)
        exec_time = time.clock() - tstart
        print(’Function {} executed in {} s’.format(func.__name__, exec_time))
        return result
    return wrapper

@clocked
def square_list(input_list):
    """ Return the square of a list"""
    return [item**2 for item in input_list]

# Make sure the function name and docstring look the same
print(’\’{}\’: {}’.format(square_list.__name__, square_list.__doc__))
square_list(range(2000000))

[Output]
’square_list’:  Return the square of a list
Function square_list executed in 0.372302 s
```

The @classmethod decorator

- We have already seen a built-in Python decorator: *@property*
- We used that to get proper encapsulation
- There is another built-in decorator one which is very useful for classes: *@classmethod*
- A classmethod is like a class attribute: you don´t need an instance to use it
- A class method can access class attributes but not instance attributes
- The main use for class methods is to provide alternate constructors

Class method

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

```
import numpy

class LabData:

  def __init__(self, times, values):
     """ Our usual constructor"""
     self.times = numpy.array(times, dtype=numpy.float64)
     self.values = numpy.array(values, dtype=numpy.float64)

  @classmethod # The classmethod decorator
  def from_file(cls, file_path): # We get the class as first argument, not self
      """ Constructor from a file"""
      print(cls)
      times, values = numpy.loadtxt(file_path, unpack=True)
      # We call the constructor of ’cls’ which is our LabData
      # This is not a ’real’ constructor, we need to return the object!
      return cls(times, values)

# We call the alternate constructor from the class itself, not from an instance!
lab_data = LabData.from_file(’snippets/data/measurements.txt’)
print(lab_data.values)

[Output]
<class ’__main__.LabData’>
[15.2 12.4 11.7 13.2]
```

Generators

Generators

- We know how to iterate over python built-in container or even create our custom iterables
- 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 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)
- Generators can be used whenever an iterable is expected

Generators first look

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

```python
""" range() is a function that returns a generator in Python 3. The list of
numbers never exists entirely, they are created ine at a time.
Note: In Python 2 range() does create the full list at the beginning.
There used to be a xrange() function for lazy generation, which is now
deprecated in Python 3. """
for i in range(4):  # generators act like iterators in for loop
    print(i)

data = [12, -1, 5]
square_data_generator = (x**2 for x in data) # generator expression!
for square_datum in square_data_generator: # again,
    print(square_datum)

[Output]
0
1
2
3
144
1
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

```
def generator_function_simple():
    print(’First call’)
    yield 1
    print(’Second call’)
    yield 2
    print(’I am about to rise a StopIteration exception...’)

gen = generator_function_simple() # A generator function returns a generator
print(next(gen)) # We stop at the first yield and get the value
print(next(gen)) # Second yield
next(gen) # The third next() will throw StopIteration

[Output]
First call
1
Second call
2
I am about to rise a StopIteration exception...
Traceback (most recent call last):
  File "snippets/generator_functions.py", line 11, in <module>
    next(gen) # The third next() will throw StopIteration
StopIteration
```

Infinite sequence generators

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

```python
# Generator function that provides infinte fibonacci numbers
def fibonacci():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

# We need to impose a stop condition externally to use it
max_n = 7
fib_numbers = []
for i, fib in enumerate(fibonacci()):
    if i >= max_n:
        break
    else:
        fib_numbers.append(fib)
print(fib_numbers)

# Another way to do that  is using ’islice’ from itertools
import itertools
# Generator expression
fib_gen = (fib for fib in itertools.islice(fibonacci(), max_n))
print(list(fib_gen))

[Output]
[0, 1, 1, 2, 3, 5, 8]
[0, 1, 1, 2, 3, 5, 8]
```

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

```python
# Here we create a lambda function and assign a name to it (ironically)
multiply = lambda x, y: x * y
# Use it
print(multiply(5, -1))

# Typical use is inside generator functions
numbers = range(10)
squares = list(map(lambda n: n**2, numbers))
print(squares)

# However, remeber that you can do the same with list comprehension
squares = [n**2 for n in numbers]
print(squares)

[Output]
-5
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```