



Closures & Generators

Walter Cazzola

Closures

- our case study
- regular expressions
- plural() via reg. expressions
- function objects
- list of patterns
- file of patterns

Generators

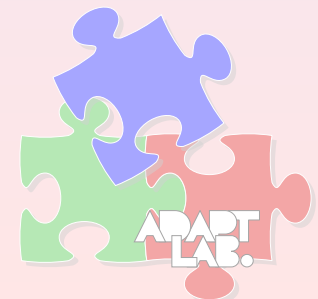
- a counter generator
- Fibonacci's Generator
- plural() via generators

References

Closures & Generators

Walter Cazzola

Dipartimento di Informatica
Università degli Studi di Milano
e-mail: cazzola@di.unimi.it
twitter: [@w_cazzola](https://twitter.com/w_cazzola)





Closures

On a Real Problem

Closures &
Generators

Walter Cazzola

Closures

our case study
regular
expressions
plural() via
reg. expressions
function objects
list of patterns
file of patterns

Generators

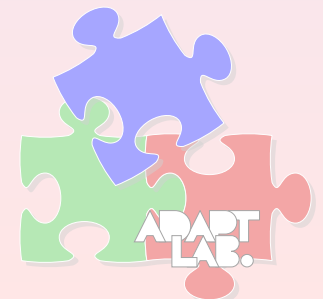
a counter
generator
Fibonacci's
Generator
plural() via
generators

References

English, from singular to plural

- if a word ends in **S**, **X**, or **Z**, add **ES**, e.g., fax becomes faxes;
- if a word ends in a **noisy H**, add **ES**, e.g., coach becomes coaches;
- if it ends in a **silent H**, just add **S**, e.g., cheetah becomes cheetahs.
- if a word ends in **Y that sounds like I**, change the **Y** to **IES**, e.g., vacancy becomes vacancies;
- if the **Y** is combined with a vowel to sound like something else, just add **S**, e.g., day becomes days;
- if all else fails, just add **S** and hope for the best.

We will design a Python module that automatically pluralizes English nouns.





Closures

Regular Expressions

Closures &
Generators

Walter Cazzola

Closures

our case study

regular
expressions

plural() via

reg. expressions

function objects

list of patterns

file of patterns

Generators

a counter
generator

Fibonacci's
Generator

plural() via
generators

References

A **Regular Expression** is a pattern to describe strings.

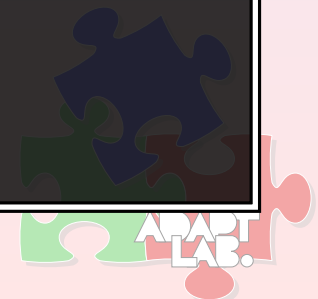
- the functions in the `re` module enables us to check if a regular expression matches a string and to return the result of the match.

Few Bytes of syntax

| | |
|----------|--|
| '.' | any character But a newline |
| '^' | the begin of the string |
| '\$' | the end of the string |
| '*', '+' | 0 (or 1) or more repetitions of the preceding RE |
| '?' | 0 or 1 repetitions of the preceding RE |
| '[]' | a set of characters |
| '()' | matching group |

RE at work

```
[22:55]cazzola@hymir:~/esercizi-pa>python3
>>> email = 'cazzola@remove_thisi.unimi.it'
>>> import re
>>> m = re.search("remove_this", email)
>>> email[:m.start()]+email[m.end():]
'cazzola@di.unimi.it'
```





Closures

Pluralizes via Regular Expressions

Closures &
Generators

Walter Cazzola

Closures

our case study

regular
expressions

plural() via
reg. expressions

function objects

list of patterns

file of patterns

Generators

a counter
generator

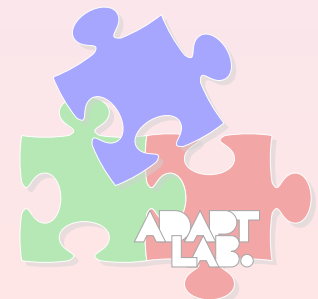
Fibonacci's
Generator

plural() via
generators

References

```
import re
def plural(noun):
    if re.search('[sxz]$', noun):
        return re.sub('$', 'es', noun)
    elif re.search('[^aeioudgkprt]h$', noun):
        return re.sub('$', 'es', noun)
    elif re.search('[^aeiou]y$', noun):
        return re.sub('y$', 'ies', noun)
    else: return noun + 's'
```

- the 1st regular expression looks for words ending by s, x or z
- the 2nd regular expression looks for words ending by a not silent h by excluding the letters that combined with it will mute the h
- the 3rd regular expression looks for words ending by a y that doesn't sound as a i similarly to the previous.





Closures

Do Some Abstraction: A List of Functions

Closures &
Generators

Walter Cazzola

Closures

our case study

regular
expressions

plural() via
reg. expressions

function objects

list of patterns

file of patterns

Generators

a counter
generator

Fibonacci's
Generator

plural() via
generators

References

To abstract we have

- to limit the number of tests to be done;
- to generalize the approach

```
import re

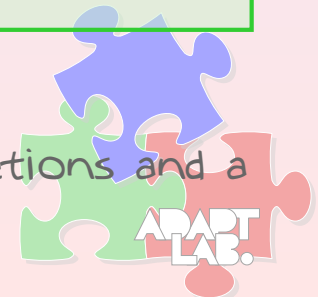
def match_sxz(noun): return re.search('[sxz]$', noun)
def apply_sxz(noun): return re.sub('$', 'es', noun)
def match_h(noun): return re.search('[^aeioudgkprt]h$', noun)
def apply_h(noun): return re.sub('$', 'es', noun)
def match_y(noun): return re.search('[^aeiou]y$', noun)
def apply_y(noun): return re.sub('y$', 'ies', noun)
def match_default(noun): return True
def apply_default(noun): return noun + 's'

rules = ((match_sxz, apply_sxz), (match_h, apply_h), (match_y, apply_y),
         (match_default, apply_default))

def plural(noun):
    for matches_rule, apply_rule in rules:
        if matches_rule(noun):
            return apply_rule(noun)
```

Advantages

- to add new rules simply means to add a couple of functions and a tuple in the rules tuple





Closures &
Generators

Walter Cazzola

Closures

our case study

regular
expressions

plural() via
reg. expressions

function objects

list of patterns

file of patterns

Generators

a counter
generator

Fibonacci's
Generator

plural() via
generators

References

Slide 6 of 11

Closures

Do Some Abstraction: A List of Patterns

To do better, we have

- to avoid to write the single functions (Boring & error-prone task)

```
import re

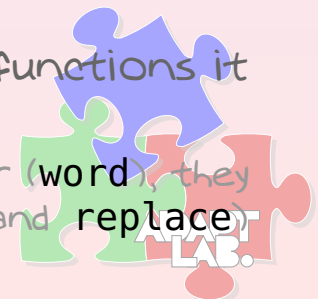
def build_match_and_apply_functions(pattern, search, replace):
    def matches_rule(word):
        return re.search(pattern, word)
    apply_rule = lambda word : \
        re.sub(search, replace, word)
    return (matches_rule, apply_rule)

patterns = ( \
    ('[sxz]$', '$', 'es'), ('^[^aeiou]h$', '$', 'es'),
    ('(qu|[^aeiou])y$', 'y$', 'ies'), ('$', '$', 's')
)

rules = [ \
    build_match_and_apply_functions(pattern, search, replace)
    for (pattern, search, replace) in patterns ]
```

The technique of binding a value within the scope definition to a value in the outside scope is named **closures**.

- It fixes the value of some variables in the body of the functions it builds:
 - Both `matches_rule` and `apply_rule` take one parameter (`word`), they act on that plus three other values (`pattern`, `search` and `replace`) which were set when the functions are built.





Closures

Do Some Abstraction: A File of Patterns

Closures &
Generators

Walter Cazzola

Closures

our case study

regular
expressions

plural() via
reg. expressions

function objects

list of patterns

file of patterns

Generators

a counter
generator

Fibonacci's
Generator

plural() via
generators

References

Separate data from code.

- By moving the patterns in a separate file.

```
[15:59]cazzola@hymir:~/esercizi-pa>cat plural-rules.txt
[sxz]$          $ es
[^aeiou dgkprt]h$ $ es
[^aeiou]y$      y$ ies
$               $ s
```

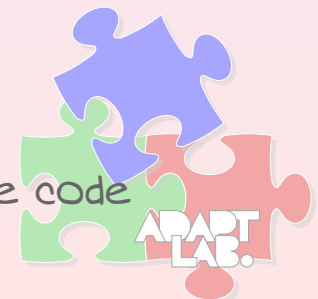
Everything is still the same But

- how is the rules list filled?

```
rules = []
with open('plural-rules.txt', encoding='utf-8') as pattern_file:
    for line in pattern_file:
        pattern, search, replace = line.split(None, 3)
        rules.append(build_match_and_apply_functions(pattern, search, replace))
```

Benefits & Drawbacks

- no need to change the code in order to add a new rule
- to read a file is slower than to hardwire the data in the code





Generators

Introduction By Example

A **generator** is a function that generates a value at a time

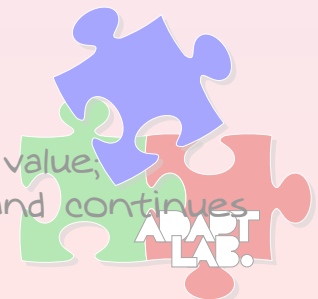
- a sort of resumable function or function with a memory

```
def make_counter(x):  
    print('entering make_counter')  
    while True:  
        yield x  
        print('incrementing x')  
        x = x + 1
```

Let look at what happens here.

```
[12:53]cazzola@hymir:~/esercizi-pa>python3  
>>> import counter  
>>> counter = counter.make_counter(2)  
>>> next(counter)  
entering make_counter  
2  
>>> next(counter)  
incrementing x  
3
```

- a call to the function initializes the generator;
- the **next()** will "synchronize" with the **yield** statement;
 - the **yield** suspends the function execution and returns a value;
 - the **next()** resumes the computation from the **yield** and continues until it reaches another **yield** or the function end.





Generators

Fibonacci's Generator

Closures &
Generators

Walter Cazzola

Closures

our case study
regular
expressions
plural() via
reg. expressions
function objects
list of patterns
file of patterns

Generators

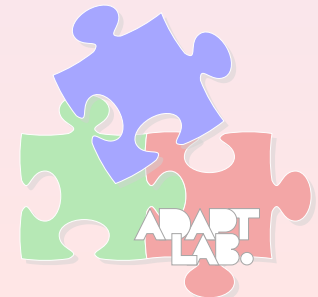
a counter
generator
Fibonacci's
Generator
plural() via
generators

References

```
def gfib(max):  
    a, b = 0, 1  
    while a < max:  
        yield a  
        a, b = b, a + b  
  
if __name__ == "__main__":  
    for n in gfib(1000):  
        print(n, end=' ')  
    print()
```

```
[15:43]cazzola@hymir:~/esercizi-pa>python3 gfib.py  
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987  
[15:52]cazzola@hymir:~/aux_work/projects/python/esercizi-pa>python3  
>>> import gfib  
>>> list(gfib.gfib(1000))  
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987]
```

- a generator can be used in a for statement, the **next()** is automatically called at each iteration
- the list constructor has a similar behavior.





Generators

Pluralizes Via Generators

Closures &
Generators

Walter Cazzola

Closures

our case study
regular
expressions
plural() via
reg. expressions
function objects
list of patterns
file of patterns

Generators

a counter
generator
Fibonacci's
Generator
plural() via
generators

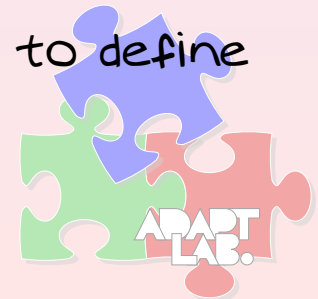
References

```
def rules(rules_filename):  
    with open(rules_filename, encoding='utf-8') as pattern_file:  
        for line in pattern_file:  
            pattern, search, replace = line.split(None, 3)  
            yield build_match_and_apply_functions(pattern, search, replace)  
  
def plural(noun, rules_filename='plural-rules.txt'):  
    for matches_rule, apply_rule in rules(rules_filename):  
        if matches_rule(noun):  
            return apply_rule(noun)  
    raise ValueError('no matching rule for {0}'.format(noun))
```

Benefits & Drawbacks

- shorter start-up time (it just reads a row not the whole file)
 lazy approach
- performance losses (every call to plural() reopens the file and reads it from the beginning again).

To get the Benefits from Both approaches you need to define your own iterator.





References

Closures &
Generators

Walter Cazzola

Closures

our case study
regular
expressions
plural() via
reg. expressions
function objects
list of patterns
file of patterns

Generators

a counter
generator
Fibonacci's
Generator
plural() via
generators

References

- ▶ Jennifer Campbell, Paul Gries, Jason Montojo, and Greg Wilson.
Practical Programming: An Introduction to Computer Science Using Python.
The Pragmatic Bookshelf, second edition, 2009.
- ▶ Mark Lutz.
Learning Python.
O'Reilly, third edition, November 2001.
- ▶ Mark Pilgrim.
Dive into Python 3.
Apress*, 2009.

