pexpect and testing

1. Pexpect
   - dialogue with an interactive program
   - pexpect commands
   - playing a number guessing game
2. Pattern Matching and Regular Expressions
   - using the `re` module
   - common regular expression symbols
   - groups of regular expressions
3. Testing Computer Algebra Systems
   - testing software
   - testing if SageMath is present
   - running the test on SymPy in SageMath

MCS 507 Lecture 16
Mathematical, Statistical and Scientific Software
Jan Verschelde, 27 September 2023

pexpect and testing

1. Pexpect
   - dialogue with an interactive program
   - pexpect commands
   - playing a number guessing game
2. Pattern Matching and Regular Expressions
   - using the `re` module
   - common regular expression symbols
   - groups of regular expressions
3. Testing Computer Algebra Systems
   - testing software
   - testing if SageMath is present
   - running the test on SymPy in SageMath

dialogue with an interactive program

Assume the following situation:

1. We have only the executable version of a program.
2. The program has a command line interpreter interface.

How to interface?

Pexpect is a pure Python module for spawning child application:
the child application can be controlled as in typing commands.

It works as Expect, an extension of the Tcl scripting language.

Pexpect is free and open source software, easy to install.
The current release 4.8, by Noah Spurrier and contributors.

Our first `hello world` experience with Pexpect:
launch a `python3.6` session from the python 3.7 interpreter.

launching python3.6

```
$ python3 pexpect_hello_python3_6.py
spawning python3.6 session ...

Python 3.6.0 (v3.6.0:41df79263a11, Dec 22 2016, 17:23:13)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more i
>>> import sys
>>> print(sys.version)
3.6.0 (v3.6.0:41df79263a11, Dec 22 2016, 17:23:13)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)]
>>> exit()

Bye, script ends.
$
```

- The user of the python3 session types the `exit()`.
- Then the code in the python3.6 replies with `Bye`.

the script pexpect_hello_python3_6.py

```
print('spawning python3.6 session ...\n')
import pexpect
CHILD = pexpect.spawn('/usr/bin/env python3.6')
# search the stream for the prompt
CHILD.expect('>>> ')
# print all data in the stream before the prompt
print(CHILD.before.decode('UTF-8'), end='')
# print the data that was matched
AFT = CHILD.after.decode()
print(AFT, end='')
# give control to interactive user
CHILD.interact()
# to end the child process before the parent ends
CHILD.close()
print('\nBye, script ends.')
```

pexpect and testing

1. Pexpect
   - dialogue with an interactive program
   - pexpect commands
   - playing a number guessing game
2. Pattern Matching and Regular Expressions
   - using the `re` module
   - common regular expression symbols
   - groups of regular expressions
3. Testing Computer Algebra Systems
   - testing software
   - testing if SageMath is present
   - running the test on SymPy in SageMath

pexpect commands

The class spawn of the module pexpect is the main interface:

1. The syntax for spawning a process is

```
>>> import pexpect
>>> child = pexpect.spawn(command)
```

2. At the creation of an object of the class spawn, a string is passed which contains the command to launch a program.

3. A child process is spawned and through this object we can interact with the application spawned.

interacting with an application

To interact with the application, we apply the method expect to the object child of the class spawn.

The main argument of the expect method is a pattern, expressed as a regular expression (as defined by the `re` module).

After a match is found, the attributes 'before', 'after', and 'match' of the spawned object `child` are set:

`child.before`: all data before the match

`child.after`: the data that was matched

`child.match`: the re.MatchObject used in the re match

The interact gives control of the child process to the interactive user.

matching the end of line

Pexpect reads one character from the stream at a time,
but often it is convenient to process data line after line.

If the spawned process is called `child`,
then to look for the end of the line:

```
child.expect('\r\n')
```

Why so? Each line in TTY devices ends with a CR/LF combination.

pexpect and testing

1. Pexpect
   - dialogue with an interactive program
   - pexpect commands
   - playing a number guessing game
2. Pattern Matching and Regular Expressions
   - using the `re` module
   - common regular expression symbols
   - groups of regular expressions
3. Testing Computer Algebra Systems
   - testing software
   - testing if SageMath is present
   - running the test on SymPy in SageMath

a simple number guessing game

The script `game_oracle_number.py`:

```
from random import randint
SECRET = randint(0, 9)
print('Welcome to our number guessing game!')
while True:
    GUESS = int(input('give a natural number in [0, 9] : '))
    if GUESS == SECRET:
        break
    if GUESS < SECRET:
        print('-> your guess is too low')
    else:
        print('-> your guess is too high')
print('Congratulations!  You found the secret.')
```

the script pexpect_interactive_game.py

```
print('spawning the oracle ...\n')
import pexpect
CHILD = pexpect.spawn\
    ('/usr/bin/env python game_oracle_number.py')
# search the stream for the prompt
CHILD.expect(': ')
# print all data in the stream before the prompt
print(CHILD.before.decode(), end='')
# print the data that was matched
print(CHILD.after.decode(), end = ''),
# give control to interactive user
try:
    CHILD.interact()
except:
    print('Child exited, game over.')
print('\nBye, script ends.')
```

playing the game interactively

```
$ python3 pexpect_interactive_game.py
spawning the oracle ...

Welcome to our number guessing game!
give a natural number in [0, 9] : 5
-> your guess is too high
give a natural number in [0, 9] : 2
-> your guess is too low
give a natural number in [0, 9] : 4
Congratulations!  You found the secret.

Bye, script ends.
$
```

the script pexpect_guessing_game.py

```
print('spawning the oracle ...\n')
import pexpect
CHILD = pexpect.spawn\
    ('/usr/bin/env python game_oracle_number.py')
for i in range(10):
    CHILD.expect(':')
    CHILD.sendline(str(i))
    CHILD.read(6)
    REPLY = CHILD.after.decode()
    print('reply of oracle', REPLY)
    if REPLY[-2:] == 'Co':
        print('We guessed the secret!')
        break
CHILD.close()
print('\nBye, script ends.')
```

script plays the game

```
$ python3 pexpect_guessing_game.py
spawning the oracle ...

reply of oracle  0
->
reply of oracle  1
->
reply of oracle  2
Co
We guessed the secret!

Bye, script ends.
$
```

two exercises

Exercises:

1. Modify the `pexpect_guessing_game.py`
   so the program takes the feedback of the oracle into account
   and applies a bisection strategy to guess the secret.
2. Consider MPSolve of Lecture 6.
   - Use Pexpect to design an interface to MPSolve.
   - For some random polynomials, compare the output of MPSolve with the numerical root finder `roots` of numpy.

automated testing with Pexpect

Pexpect makes Python a better tool for controlling other applications.

Interactive applications such as ssh, passwd, etc. can be automated.

Another application is to automate setup scripts for duplicating software package installations on different servers.

Many computer algebra systems use the same syntax for mathematical expressions so one can automate testing.

It can also be used to control web applications via `lynx`, or some other text-based web browser.

pexpect and testing

1. Pexpect
   - dialogue with an interactive program
   - pexpect commands
   - playing a number guessing game
2. Pattern Matching and Regular Expressions
   - using the `re` module
   - common regular expression symbols
   - groups of regular expressions
3. Testing Computer Algebra Systems
   - testing software
   - testing if SageMath is present
   - running the test on SymPy in SageMath

regular expressions — using the `re` module

Manipulating text and strings is an important task:

- parse to ensure entered data is correct,
- search through confidential data: use program.

Suppose `answer` contains the answers
to a yes or no question.

Acceptable *yes* answers:

1. `y` or `yes`
2. `Y` or `Yes`

Testing all these cases is tedious.

Support for **r**egular **e**xpressions:

```
>>> import re
```

`re` is a standard library module.

matching short and long answers

```
>>> import re
>>> short = 'y'; long = 'Yes'
>>> re.match('y',short) != None
True
>>> re.match('y',long) != None
False
>>> re.match('y|Y',long) != None
True
>>> re.match('y|Y',long)
<_sre.SRE_Match object at 0x5cb10>
>>> re.match('y|Y',long).group()
'Y'
>>> re.match('y|Y',short).group()
'y'
```

matching strings with `match()`

The function `match()` in the `re` module:

```
>>> re.match( < pattern > , < string > )
```

If the `string` does not match the `pattern`,
then `None` is returned.

If the `string` matches the `pattern`,
then a match object is returned.

```
>>> re.match('he','hello')
<_sre.SRE_Match object at 0x5cb10>
>>> re.match('hi','hello') == None
True
```

the `group()` method

What can we do with the match object?

```
>>> re.match('he','hello')
<_sre.SRE_Match object at 0x5cb10>
>>> _.group()
'he'
```

After a successful match, `group()` returns
that part of the pattern that matches the string.

the methods `search()` and `match()`

The match only works from the start:

```
>>> re.match('ell','hello') == None
True
```

Looking for the first occurrence of the pattern in the string, with `search()`:

```
>>> re.search('ell','hello')
<_sre.SRE_Match object at 0x5cb10>
>>> _.group()
'ell'
```

pexpect and testing

1. Pexpect
   - dialogue with an interactive program
   - pexpect commands
   - playing a number guessing game
2. Pattern Matching and Regular Expressions
   - using the `re` module
   - common regular expression symbols
   - groups of regular expressions
3. Testing Computer Algebra Systems
   - testing software
   - testing if SageMath is present
   - running the test on SymPy in SageMath

regular expressions

pattern	strings matched
`literal`	strings starting with `literal`
`re1|re2`	strings starting with `re1` or `re2`

```
>>> from time import ctime
>>> now = ctime()
>>> now
'Mon Nov 11 07:19:44 2013'
>>> p = ...
'\w{3}\s\w{3}\s\d{2}\s\d{2}:\d{2}:\d{2}\s\d{4}'
>>> re.match(p,now) != None
True
```

pattern	strings matched
`\w`	any alphanumeric character, same as `[A-Za-z]`
`\d`	any decimal digit, same as `[0-9]`
`\s`	any whitespace character
`re{`*n*`}`	*n* occurrences of `re`

matching 0 or 1 occurrences

Allowing `Ms.`, `Mr.`, `Mrs.`, with or without the `.` (dot)

```
>>> title = 'Mr?s?\.? '
```

- `?` matches 0 or 1 occurrences
- `.` matches any character
- `\.` matches the dot `.`

```
>>> re.match(title,'Ms ') != None
True
>>> re.match(title,'Ms. ') != None
True
>>> re.match(title,'Miss ') != None
False
>>> re.match(title,'Mr') != None
False
>>> re.match(title,'Mr ') != None
True
>>> re.match(title,'M ') != None
True
```

character classes

Match with specific characters.

A name has to start with upper case:

```
>>> name = '[A-Z][a-z]*'
>>> G = 'Guido van Rossum'
>>> re.match(name,G)
>>> _.group()
'Guido'
>>> g = 'guido'
>>> re.match(name,g) == None
True
```

pexpect and testing

1. Pexpect
   - dialogue with an interactive program
   - pexpect commands
   - playing a number guessing game
2. Pattern Matching and Regular Expressions
   - using the `re` module
   - common regular expression symbols
   - groups of regular expressions
3. Testing Computer Algebra Systems
   - testing software
   - testing if SageMath is present
   - running the test on SymPy in SageMath

the `groups()` method

Groups of regular expressions are designated with parenthesis, between `(` and `)`.

Syntax:

```
< pattern > = ( < group1 > ) ( < group2 > )
m = re.match( < pattern > , < string > )
if m != None: m.groups()
```

After a successful match, groups() returns a tuple of those parts of the string that matched the pattern.

extracting hours, seconds, minutes

```
>>> from time import ctime
>>> now = ctime()
>>> now
'Mon Nov 11 07:19:44 2013'
>>> t = now.split(' ')[3]
>>> t
'07:32:07'
>>> format = '(\d\d):(\d\d):(\d\d)'
>>> m = re.match(format,t)
>>> m.groups()
('07', '32', '07')
>>> (hours, minutes, seconds) = _
>>> minutes
'32'
```

pexpect and testing

1. Pexpect
   - dialogue with an interactive program
   - pexpect commands
   - playing a number guessing game
2. Pattern Matching and Regular Expressions
   - using the `re` module
   - common regular expression symbols
   - groups of regular expressions
3. Testing Computer Algebra Systems
   - testing software
   - testing if SageMath is present
   - running the test on SymPy in SageMath

testing software

We distinguish between

- test to fail, e.g.: careless user, special cases, stress test, ...
- test to pass, e.g.: meets specifications, regression test.

In this lecture we will test to pass.

SageMath contains several computer algebra systems:
Maxima, PARI/GP, GAP, Singular, SymPy.

How do we test the systems systematically and automatically?

- prepare a test suite of problems,
- apply Pexpect to run the test scripts.

testing computer algebra

Michael J. Wester: **A Critique of the Mathematical Abilities of CA Systems**, pages 25-60, published as chapter 3 in *Computer Algebra Systems. A Practical Guide*, edited by Michael J. Wester, Wiley 1999.

This chapter contains 20 pages with tables of 542 test problems. Consider problems C1 and C2 on numbers:

$$\begin{aligned} 50! &= 30414093201713378043612608166064768844377\ldots \\ &\quad \ldots 641568960512000000000000 \\ &= 2^{47}3^{22}5^{12}7^{8}11^{4}13^{3}17^{2}19^{2}23^{2}\ 29\ 31\ 37\ 41\ 43\ 47 \end{aligned}$$

We will verify the results by visual inspection and *assert* that

1. 50! is a number of 65 decimal places long; and
2. 50! has 15 prime factors.

a script with `sympy`

```
import sympy as sp
N = sp.factorial(50)
print('50! =', N)
print('50! has', len(str(N)), 'decimal places')
F = sp.factorint(N)
K = list(F.keys())
K.sort()
L = [str(x)+'^'+str(F[x]) for x in K]
S = ' '.join(L)
print('50! =', S)
print('50! has', len(F.keys()), 'prime factors')
```

running the test script

The running of the script (with edited output is below):

```
$ python3 factorial50_sympy.py
50! = 30414093201713378043612608166064768844377 \
      641568960512000000000000
50! has 65 decimal places
50! = 2^47 3^22 5^12 7^8 11^4 13^3 17^2 19^2 23^2 \
      29^1 31^1 37^1 41^1 43^1 47^1
50! has 15 prime factors
$
```

We replace the print statement by two `assert` statements:

1. 50! has 65 decimal places: `assert(len(str(N)) == 65)`
2. 50! has 15 factors: `assert(len(L) == 15)`

The `assert()` raises `AssertionError` if the condition fails.

replacing prints by `assert()` statements

```
import sympy as sp
N = sp.factorial(50)
assert(len(str(N)) == 65)
F = sp.factorint(N)
assert(len(F.keys()) == 15)
print('test passed')
```

pexpect and testing

1. Pexpect
   - dialogue with an interactive program
   - pexpect commands
   - playing a number guessing game
2. Pattern Matching and Regular Expressions
   - using the `re` module
   - common regular expression symbols
   - groups of regular expressions
3. Testing Computer Algebra Systems
   - testing software
   - testing if SageMath is present
   - running the test on SymPy in SageMath

testing if SageMath is present

Instead of running scripts with `sage` at the command line, the test program reads a script line by line and sends the lines to `sage` with Pexpect.

As a starter, let us run a script that checks whether sage is available.

```
$ python3 sage_version_test.py
version of SageMath : 8.8
$
```

the script sage_version_test.py

```
import pexpect

def sage_version():
    """
    Returns a string containing sage version
    and release date or None if sage cannot
    be found at the execution path.
    """
    try:
        child = pexpect.spawn('/usr/bin/env sage')
    except:
        return None
    child.expect('sage')
    first = child.before.decode()
    lines = first.split('version')
    return lines[1][1:4]

if __name__ == "__main__":
    print('version of SageMath :', sage_version())
```

pexpect and testing

1. Pexpect
   - dialogue with an interactive program
   - pexpect commands
   - playing a number guessing game
2. Pattern Matching and Regular Expressions
   - using the `re` module
   - common regular expression symbols
   - groups of regular expressions
3. Testing Computer Algebra Systems
   - testing software
   - testing if SageMath is present
   - running the test on SymPy in SageMath

running the test on SymPy in SageMath

We keep the original script:

```
import sympy as sp
N = sp.factorial(50)
assert(len(str(N)) == 65)
F = sp.factorint(N)
assert(len(F.keys()) == 15)
print('test passed')
```

and feed the script to SageMath with Pexpect.

Instead of running the script with `sage`,
we run the test statements one after the other in Python.

the script `factorial50_sympy_test.py`

```
import pexpect
from sage_version_test import sage_version

V = sage_version()
if(V == None):
    print('Sage not installed?')
else:
    print('version of sage :', V)
    CHILD = pexpect.spawn('/usr/bin/env sage')
    CHILD.expect('sage:')
    SCRIPT = open('factorial50_sympy_assert.py', 'r')
```

the script continued

```
while True:
    LINE = SCRIPT.readline()
    if(LINE == ""):
        break
    if LINE[0] != '#':
        CMD = LINE[:-1]
        print('executing', CMD)
        CHILD.sendline(CMD)
        CHILD.expect('sage:')
        if(CMD[:5] == 'print'):
            C = CHILD.before.decode()
            L = C.split(CMD)
            for line in L:
                print(line)
CHILD.close()
print('test passed')
```

Summary + Additional Exercises

Pexpect is an easy to use tool to control application in Python.
We consider the setup of automated testing with Pexpect.

Additional Exercises:

3. The class `Maxima` in `sage.interfaces.maxima` defines an interface to pass commands directly to maxima.
   Use Pexpect to automate the 50! test on Maxima.

4. Consider any of the remaining 540 problems of the paper of Michael J. Wester.
   Develop for your selected problem an automated test.