樹莓派, 樹莓派之學習, 樹莓派之教育

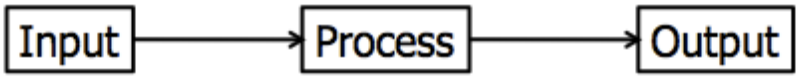
STEM 隨筆：古典力學：模擬術【小工具】七

2018-06-10 | 懸鉤子 | 發表迴響

通常熟悉教材思路模式利於掌握學習進入狀況。故此先講

輸入．處理．輸出模型：

IPO model



The **input–process–output (IPO) model**, or **input-process-output** pattern, is a widely used approach in systems analysis and software engineering for describing the structure of an information processing program or other process. Many introductory programming and systems analysis texts introduce this as the most basic structure for describing a process.[1][2][3][4]

Overview

A computer program or any other sort of process using the input-process-output model receives inputs from a user or other source, does some computations on the inputs, and returns the results of the computations.[1] In essence the system separates itself from the environment, thus defining both inputs and outputs, as one united mechanism.[5] The system would divide the work into two categories:

- A requirement from the environment (input)
- A provision for the environment (output)

In other words, such inputs may be materials, human resources, money or information, transformed into outputs, such as consumables, services, new information or money.

As a consequence, Input-Process-Output system becomes very vulnerable to misinterpretation. This is because, theoretically, it contains all the data, in regards to the environment outside the system, yet on practice, environment contains a significant variety of objects, that a system is unable to comprehend, as it exists outside systems control. As a result it is very important, to understand, where the boundary lies, between the system and the environment, which is beyond systems understanding. This is because, often various analysts, would set their own boundaries, favouring their point of view, thus creating much confusion.[6]

指明其與 IPython 之

讀取・求值・輸出循環：

Read–eval–print loop

A **Read–Eval–Print Loop** (**REPL**), also known as an **interactive toplevel** or **language shell**, is a simple, interactive computer programming environment that takes single user inputs (i.e. single expressions), evaluates them, and returns the result to the user; a program written in a REPL environment is executed piecewise. The term is most usually used to refer to programming interfaces similar to the classic Lisp machine interactive environment. Common examples include command line shells and similar environments for programming languages, and is particularly characteristic of scripting languages.[1]

Uses

As a shell, a REPL environment allows users to access relevant features of an operating system in addition to providing access to programming capabilities.

The most common use for REPLs outside of operating system shells is for instantaneous

prototyping. Other uses include mathematical calculation, creating documents that integrate scientific analysis (e.g. IPython), interactive software maintenance, benchmarking, and algorithm exploration.

A REPL can become an essential part of learning a new language as it gives quick feedback to the novice.

密切相關也。這可說是

《Jupyter Audio Basics》文章綱要哩！

——《【鼎革．革鼎】：RASPBIAN STRETCH《六之 J.3．MIR-3》》

望著

Output widgets: leveraging Jupyter’s display system

文本發呆！想著格與格間之互動？隨機而添、而抹的筆意！！

哪裡得『五色筆』

齊光祿江淹

文通詩體總雜，善於摹擬，筋力於王微，成就於謝朓。初，淹罷宣城郡，遂宿冶亭，夢一美丈夫，自稱郭璞，謂淹曰：「我有筆在卿處多年矣，可以見還。」淹探懷中，得五色筆以授之。爾後為詩，不復成語，故世傳江淹才盡。

方能寫此璀璨？？

既無下筆處，只好有請讀者

何不『**吃茶去**』？？？

《趙州禪師語錄》

（459）師問二新到：「上座曾到此間否？」云：「不曾到。」師云：「**吃茶去！**」又問那一人：「曾到此間否？」云：「曾到。」師云：「**吃茶去！**」院主問：「和尚！不曾到，教伊**吃茶去**，即且置；曾到，為什麼教伊**吃茶去**？」師云：「院主。」院主應諾。師云：「**吃茶去！**」

Output widgets: leveraging Jupyter's display system

```
import ipywidgets as widgets
```

The `Output` widget can capture and display stdout, stderr and rich output generated by IPython. You can also append output directly to an output widget, or clear it programmatically.

```
out = widgets.Output(layout={'border': '1px solid black'})
out
```

```
0 Hello world!
1 Hello world!
2 Hello world!
3 Hello world!
4 Hello world!
5 Hello world!
6 Hello world!
7 Hello world!
8 Hello world!
9 Hello world!
```

0

After the widget is created, direct output to it using a context manager. You can print text to the output area:

```
with out:
    for i in range(10):
        print(i, 'Hello world!')
```

Rich output can also be directed to the output area. Anything which displays nicely in a Jupyter notebook will also display well in the `Output` widget.

```
from IPython.display import YouTubeVideo
with out:
    display(YouTubeVideo('eWzY2nGfkXk'))
```

We can even display complex mimetypes, such as nested widgets, in an output widget.

```
with out:
    display(widgets.IntSlider())
```

We can also append outputs to the output widget directly with the convenience methods `append_stdout`, `append_stderr`, or `append_display_data`.

```
out = widgets.Output(layout={'border': '1px solid black'})
out.append_stdout('Output appended with append_stdout')
out.append_display_data(YouTubeVideo('eWzY2nGfkXk'))
out
```

Note that `append_display_data` cannot currently be used to display widgets. The status of this bug is tracked in this issue.

We can clear the output by either using `IPython.display.clear_output` within the context manager, or we can call the widget's `clear_output` method directly.

```
out.clear_output()
```

`clear_output` supports the keyword argument `wait`. With this set to `True`, the widget contents are not cleared immediately. Instead, they are cleared the next time the widget receives something to display. This can be useful when replacing content in the output widget: it allows for smoother transitions by avoiding a jarring resize of the widget following the call to `clear_output`.

Finally, we can use an output widget to capture all the output produced by a function using the `capture` decorator.

```
@out.capture()
def function_with_captured_output():
    print('This goes into the output widget')
    raise Exception('As does this')

function_with_captured_output()
```

`out.capture` supports the keyword argument `clear_output`. Setting this to `True` will clear the output widget every time the function is invoked, so that you only see the output of the last invocation. With `clear_output` set to `True`, you can also pass a `wait=True` argument to only clear the output once new output is available. Of course, you can also manually clear the output any time as well.

```
out.clear_output()
```

Output widgets as the foundation for interact

The output widget forms the basis of how interact and related methods are implemented. It can also be used by itself to create rich layouts with widgets and code output. One simple way to customize how an interact UI looks is to use the `interactive_output` function to hook controls up to a function whose output is captured in the returned output widget. In the next example, we stack the controls vertically and then put the output of the function to the right.

```
a = widgets.IntSlider(description='a')
b = widgets.IntSlider(description='b')
c = widgets.IntSlider(description='c')
def f(a, b, c):
    print('{}*{}*{}={}'.format(a, b, c, a*b*c))

out = widgets.interactive_output(f, {'a': a, 'b': b, 'c': c})

widgets.HBox([widgets.VBox([a, b, c]), out])
```

Debugging errors in callbacks with the output widget

On some platforms, like JupyterLab, output generated by widget callbacks (for instance, functions attached to the `.observe` method on widget traits, or to the `.on_click` method on button widgets) are not displayed anywhere. Even on other platforms, it is unclear what cell this output should appear in. This can make debugging errors in callback functions more challenging.

An effective tool for accessing the output of widget callbacks is to decorate the callback with an output widget's capture method. You can then display the widget in a new cell to see the callback output.

```
debug_view = widgets.Output(layout={'border': '1px solid black'})

@debug_view.capture(clear_output=True)
def bad_callback(event):
    print('This is about to explode')
    return 1.0 / 0.0

button = widgets.Button(
    description='click me to raise an exception',
    layout={'width': '300px'}
)
button.on_click(bad_callback)
button
```

click me to raise an exception

```
debug_view
```

```
This is about to explode

---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
/usr/local/lib/python3.5/dist-packages/ipywidgets/widgets/widget_output.py in inner(*args, **kwargs)
     99                     self.clear_output(*clear_args, **clear_kwargs)
    100                 with self:
--> 101                     return func(*args, **kwargs)
    102             return inner
    103         return capture_decorator

<ipython-input-11-d8bd74d19e7f> in bad_callback(event)
      4 def bad_callback(event):
      5     print('This is about to explode')
----> 6     return 1.0 / 0.0
      7
      8 button = widgets.Button(

ZeroDivisionError: float division by zero
```

Integrating output widgets with the logging module

While using the `.capture` decorator works well for understanding and debugging single callbacks, it does not scale to larger applications. Typically, in larger applications, one might use the logging module to print information on the status of the program. However, in the case of widget applications, it is unclear where the logging output should go.

A useful pattern is to create a custom handler that redirects logs to an output widget. The output widget can then be displayed in a new cell to monitor the application while it runs.

```
import ipywidgets as widgets
import logging

class OutputWidgetHandler(logging.Handler):
    """ Custom logging handler sending logs to an output widget """

    def __init__(self, *args, **kwargs):
        super(OutputWidgetHandler, self).__init__(*args, **kwargs)
        layout = {
            'width': '100%',
            'height': '160px',
            'border': '1px solid black'
        }
        self.out = widgets.Output(layout=layout)

    def emit(self, record):
        """ Overload of logging.Handler method """
        formatted_record = self.format(record)
        new_output = {
            'name': 'stdout',
            'output_type': 'stream',
            'text': formatted_record+'\n'
        }
        self.out.outputs = (new_output, ) + self.out.outputs

    def show_logs(self):
        """ Show the logs """
        display(self.out)

    def clear_logs(self):
        """ Clear the current logs """
        self.out.clear_output()


logger = logging.getLogger(__name__)
handler = OutputWidgetHandler()
handler.setFormatter(logging.Formatter('%(asctime)s  - [%(levelname)s] %(message)s'))
logger.addHandler(handler)
logger.setLevel(logging.INFO)
```

```
handler.show_logs()
```

```
2018-06-08 16:36:34,695  - [ERROR] An error occurred!
Traceback (most recent call last):
  File "<ipython-input-15-43aa6f6a7e65>", line 6, in <module>
    1.0/0.0
ZeroDivisionError: float division by zero
2018-06-08 16:36:34,688  - [INFO] About to try something dangerous...
2018-06-08 16:36:34,682  - [INFO] Starting program
```

```
handler.clear_logs()
logger.info('Starting program')

try:
    logger.info('About to try something dangerous...')
    1.0/0.0
except Exception as e:
    logger.exception('An error occurred!')
```

Interacting with output widgets from background threads

Jupyter's `display` mechanism can be counter-intuitive when displaying output produced by background threads. A background thread's output is printed to whatever cell the main thread is currently writing to. To see this directly, create a thread that repeatedly prints to standard out:

```
import threading
import time

def run():
    for i in itertools.count(0):
        time.sleep(1)
        print('output from background {}'.format(i))

t = threading.Thread(target=run)
t.start()
```

This always prints in the currently active cell, not the cell that started the background thread.

This can lead to surprising behaviour in output widgets. During the time in which output is captured by the output widget, *any* output generated in the notebook, regardless of thread, will go into the output widget.

The best way to avoid surprises is to *never* use an output widget's context manager in a context where multiple threads generate output. Instead, we can pass an output widget to the function executing in a thread, and use `append_display_data()`, `append_stdout()`, or `append_stderr()` methods to append displayable output to the output widget.

```
import threading
from IPython.display import display, HTML
import ipywidgets as widgets
import time

def thread_func(something, out):
    for i in range(1, 5):
        time.sleep(0.3)
        out.append_stdout('{} {} {}\n'.format(i, '**'*i, something))
    out.append_display_data(HTML("<em>All done!</em>"))

display('Display in main thread')
out = widgets.Output()
# Now the key: the container is displayed (while empty) in the main thread
display(out)

thread = threading.Thread(
    target=thread_func,
    args=("some text", out))
thread.start()
```

```
'Display in main thread'
```

```
1 ** some text
2 **** some text
3 ****** some text
4 ******** some text
```

All done!