LECTURE 7

NOTEBOOKS; VARIADIC FUNCTIONS

MCS 275 Spring 2021

Emily Dumas

LECTURE 7: NOTEBOOKS; VARIADIC FUNCTIONS

Course bulletins:

- Project 1 posted. Deadline 6pm CST on Fri Feb 5.
- Project 1 autograder opens on Monday.
- Work on Worksheet 3.

PYTHON INTERFACES

- **REPL** -- Run one command at a time. Result is printed. Code is lost when you exit.
- **Script mode** -- Run all commands in a file. Nothing printed unless explicitly requested (e.g. `print(...)`). Code saved in a file (by design).

PYTHON INTERFACES

- **REPL** -- Run one command at a time. Result is printed. Code is lost when you exit.
- **Notebook** -- Create and run groups of commands (cells) in a browser interface. Last result is printed. Can mix code and formatted text. Can save to a file.
- **Script mode** -- Run all commands in a file. Nothing printed unless explicitly requested (e.g. `print(...)`). Code saved in a file (by design).

WHAT NOTEBOOKS LOOK LIKE

3. Cipher class hierarchy

Build a module `encoders` (in `encoders.py`) containing classes for simple ciphers (or codes; ways of obscuring the contents of a string that can be undone later by the intended recipient).

There should be a base class `BaseEncoder` that has two methods:

- `encode(self,text)` : Returns the string `text` unchanged. Subclasses will alter this behavior.
- `decode(self,text)` : Returns the string `text` unchanged. Subclasses will alter this behavior.

It should be the case that `obj.decode(obj.encode(s)) == s` is true for any string `s`, and for any object `obj` that is an instance of `BaseEncoder` or subclass thereof.

Then, build subclasses of `BaseEncoder` that implement encoding and decoding by different ciphers, including:

- `RotateEncoder` : Encoding rotates letters in the alphabet forward by a certain number of steps, e.g. so rotation by 5 turns "a" into "f" and "z" into "e" (because we wrap around when we reach the end of the alphabet). No transformation is applied to characters other than capital and lower case letters. Constructor accepts an integer, specifying the number of steps to rotate.
- `Rot13Encoder` : A subclass of `RotateEncoder` that fixes the steps at 13, so that encoding and decoding are the same operation.
- `SubstitutionEncoder` : The constructor accepts two arguments, `pre` and `post`. The string `pre` is a list of characters to be replaced when encoding, and string `post` indicates the things to replace them with. For example, using `pre="abcd"` and `post="1j4e"` would mean that "a" is supposed to be replaced by "1", "b" by "j", "c" by "4", and so on.
  - Be careful writing the encoder so that you don't replace things twice. For example `pre="abc"` and `post="bca"` should encode "banana" to "cbnbnb", and *not* "ccncnc".
  - You can assume that `pre` and `post` contain the same characters but in a different order. If that's not the case, then it would be impossible to ensure that decoding after encoding always gives the original text back again.

You can find some test code below. The test code assumes all of the classes are in the global scope.

In [ ]:
```
E = RotateEncoder(5)
s = E.encode("Hello world!") # Mjqqt btwqi!
print(s) # Mjqqt btwqi!
print(E.decode(s)) # Hello world!

F = SubstitutionEncoder("lmno","nolm")
s = F.encode("Hello everyone!")
print(s) # Hennm everymle!
print(F.decode(s)) # Hello everyone!
```

MCS 275 uses notebooks for quizzes, worksheets, and project descriptions, so you've seen these before. But you usually see a version converted to HTML.

WHAT NOTEBOOKS LOOK LIKE

3. Cipher class hierarchy

Build a module `encoders` (in `encoders.py`) containing classes for simple ciphers (or codes; ways of obscuring the contents of a string that can be undone later by the intended recipient).

There should be a base class `BaseEncoder` that has two methods:

- `encode(self,text)` : Returns the string `text` unchanged. Subclasses will alter this behavior.
- `decode(self,text)` : Returns the string `text` unchanged. Subclasses will alter this behavior.

It should be the case that `obj.decode(obj.encode(s)) == s` is true for any string `s`, and for any object `obj` that is an instance of `BaseEncoder` or subclass thereof.

Then, build subclasses of `BaseEncoder` that implement encoding and decoding by different ciphers, including:

- `RotateEncoder` : Encoding rotates letters in the alphabet forward by a certain number of steps, e.g. so rotation by 5 turns "a" into "f" and "z" into "e" (because we wrap around when we reach the end of the alphabet). No transformation is applied to characters other than capital and lower case letters. Constructor accepts an integer, specifying the number of steps to rotate.
- `Rot13Encoder` : A subclass of `RotateEncoder` that fixes the steps at 13, so that encoding and decoding are the same operation.
- `SubstitutionEncoder` : The constructor accepts two arguments, `pre` and `post`. The string `pre` is a list of characters to be replaced when encoding, and string `post` indicates the things to replace them with. For example, using `pre="abcd"` and `post="1j4e"` would mean that "a" is supposed to be replaced by "1", "b" by "j", "c" by "4", and so on.
  - Be careful writing the encoder so that you don't replace things twice. For example `pre="abc"` and `post="bca"` should encode "banana" to "cbnbnb", and *not* "ccncnc".
  - You can assume that `pre` and `post` contain the same characters but in a different order. If that's not the case, then it would be impossible to ensure that decoding after encoding always gives the original text back again.

Cell of formatted text

You can find some test code below. The test code assumes all of the classes are in the global scope.

In [ ]:
```
E = RotateEncoder(5)
s = E.encode("Hello world!") # Mjqqt btwqi!
print(s) # Mjqqt btwqi!
print(E.decode(s)) # Hello world!

F = SubstitutionEncoder("lmno","nolm")
s = F.encode("Hello everyone!")
print(s) # Hennm everymle!
print(F.decode(s)) # Hello everyone!
```

MCS 275 uses notebooks for quizzes, worksheets, and project descriptions, so you've seen these before. But you usually see a version converted to HTML.

WHAT NOTEBOOKS LOOK LIKE

3. Cipher class hierarchy

Build a module `encoders` (in `encoders.py`) containing classes for simple ciphers (or codes; ways of obscuring the contents of a string that can be undone later by the intended recipient).

There should be a base class `BaseEncoder` that has two methods:

- `encode(self,text)` : Returns the string `text` unchanged. Subclasses will alter this behavior.
- `decode(self,text)` : Returns the string `text` unchanged. Subclasses will alter this behavior.

It should be the case that `obj.decode(obj.encode(s)) == s` is true for any string `s`, and for any object `obj` that is an instance of `BaseEncoder` or subclass thereof.

Then, build subclasses of `BaseEncoder` that implement encoding and decoding by different ciphers, including:

- `RotateEncoder` : Encoding rotates letters in the alphabet forward by a certain number of steps, e.g. so rotation by 5 turns "a" into "f" and "z" into "e" (because we wrap around when we reach the end of the alphabet). No transformation is applied to characters other than capital and lower case letters. Constructor accepts an integer, specifying the number of steps to rotate.
- `Rot13Encoder` : A subclass of `RotateEncoder` that fixes the steps at 13, so that encoding and decoding are the same operation.
- `SubstitutionEncoder` : The constructor accepts two arguments, `pre` and `post`. The string `pre` is a list of characters to be replaced when encoding, and string `post` indicates the things to replace them with. For example, using `pre="abcd"` and `post="1j4e"` would mean that "a" is supposed to be replaced by "1", "b" by "j", "c" by "4", and so on.
  - Be careful writing the encoder so that you don't replace things twice. For example `pre="abc"` and `post="bca"` should encode "banana" to "cbnbnb", and *not* "ccncnc".
  - You can assume that `pre` and `post` contain the same characters but in a different order. If that's not the case, then it would be impossible to ensure that decoding after encoding always gives the original text back again.

You can find some test code below. The test code assumes all of the classes are in the global scope.

In [ ]:
```
E = RotateEncoder(5)
s = E.encode("Hello world!") # Mjqqt btwqi!
print(s) # Mjqqt btwqi!
print(E.decode(s)) # Hello world!

F = SubstitutionEncoder("lmno","nolm")
s = F.encode("Hello everyone!")
print(s) # Hennm everymle!
print(F.decode(s)) # Hello everyone!
```

Cell of code

MCS 275 uses notebooks for quizzes, worksheets, and project descriptions, so you've seen these before. But you usually see a version converted to HTML.

HOW TO USE NOTEBOOKS

Several options:

- **Google Colab** -- Web tool to create, edit, run notebooks. Need a Google account. Can save or download notebooks.
- Other online provider, e.g. Kaggle, CoCalc
- **Jupyter/iPython** -- Software you install locally to create, edit, run notebooks. Browser shows the UI.
- **VS Code** -- Has an extension for handling notebook files.

JUPYTER, IPYTHON, IPYNB

Jupyter is software that provides a notebook interface to various languages.

iPython adds Python support to Jupyter.

Often, installing Jupyter will also install iPython.

Jupyter saves notebooks in `.ipynb` format, which most notebook software supports.

JUPYTER INSTALL INSTRUCTIONS

Most users can install Jupyter and iPython using pip:

```
python -m pip install jupyter
```

Then run the interface with:

```
python -m jupyter notebook
```

Of course, you need to replace `python` with your interpreter name.

USING COLAB / JUPYTER

A few of the many keyboard shortcuts:

- shift-enter -- run the current cell
- escape -- switch from cell editing to navigation
- a -- in nav mode, add a new cell ABOVE this one
- b -- in nav mode, add a new cell BELOW this one
- dd -- in Jupyter, in nav mode, delete current cell (colab has a delete button, and a different shortcut)
- m -- in Jupyter, in nav mode, make current cell a Markdown (text) cell

MARKDOWN

Text cells (Colab) or markdown cells (Jupyter) contain formatted text. When editing, formatting is specified with a language called Markdown.

```
# Heading level 1
## Heading level 2
### Heading level 3

* Bullet list item
* Another bullet list item

1. Numbered list item
1. Another numbered list item

Links: [text to display](https://example.com)
```

VARIADIC FUNCTIONS

A function is **variadic** if it can accept a variable number of arguments. This is general CS terminology.

Python supports these. The syntax

```
def f(a,b,*args):
```

means that the first argument goes into variable `a`, the second into variable `b`, and any other arguments are put into a tuple which is assigned to `args`

VARIADICS AND KEYWORD ARGUMENTS

The syntax

```
def f(a,b,**kwargs):
```

or

```
def f(a,b,*args,**kwargs):
```

puts extra keyword arguments into a dictionary called `kwargs`.

It is traditional to use the names `args` and `kwargs`, but it is not required.

ARGUMENT UNPACKING

Take arguments from a list or tuple:

```
L = [6,11,16]
f(1,*L) # calls f(1,6,11,16)
```

Take keyword arguments from a dict:

```
d = { "mcs275": "fun", "x": 42 }
f(1,z=0,**d) # calls f(1,z=0,mcs275="fun",x=42)
```

Think of * as "remove the brackets", and ** as "remove the curly braces".

APPLICATION

Including `...`,`*args`,`*kwargs` is especially useful on higher-order functions that accept a function as an argument, and which accept arguments on behalf of that function.

Typically you don't know in advance how many extra arguments to accept, so a variadic function is needed.

REFERENCES

- Google Colab offers notebook creation, editing, execution (can use netid@uic.edu google account).
- Some other online services allowing free use of Python notebooks: Kaggle, CoCalc
- See *Lutz*, Chapter 18 for more about function arguments (including variadic functions).
- *Beazley & Jones*, Chapter 7 has examples of variadic functions.
- A Markdown guide from GitHub.

REVISION HISTORY

- 2021-01-26 Initial publication