impracticalpythonprojects

Release 0.23.0

Jose A. Lerma III

Aug 14, 2019

MODULE REFERENCE

Example implementations of the practice and challenge projects in Impractical Python Projects. Alternative answers to practice projects and supporting files can be found at the official GitHub page.

It's a fantastic intermediate level book that has truly impractical (but fun) projects. It's a great way to get tricked into learning new conventions, techniques, and modules.

My original python-tutorials repository is already very nested, so these will be easier to find and review here; however, the original repository still has relevant information about configuring a Python environment/IDE.

Bonus content includes Google style docstrings (such wow), main functions (so standard), pip requirements files (so helpful), and test files (**not** punny at all).

CHAPTER

ONE

SRC

1.1 src package

1.1.1 Subpackages

src.ch01 package

Subpackages

src.ch01.challenge package

Submodules

src.ch01.challenge.c1_foreign_bar_chart module

Return letter 'bar chart' of a non-English sentence.

`src.ch01.challenge.c1_foreign_bar_chart.`**`add_keys_to_dict`**(*dictionary: dict*) → dict
Add keys to dictionary.

Check keys of a letter dictionary and add missing letters.

Parameters **`dictionary`** (*dict*) – Dictionary to check keys of.

Returns Dictionary with `string.ascii_lowercase` as keys.

Raises **`TypeError`** – If **dictionary** is not a `dict`.

`src.ch01.challenge.c1_foreign_bar_chart.`**`foreign_freq_analysis`**(*sentence: str*) → dict
Wrap freq_analysis and add_keys_to_dict.

Passes given sentence through *`freq_analysis()`* then *`add_keys_to_dict()`* to fill in missing keys.

Parameters **`sentence`** (*str*) – String to count letters of.

Returns Dictionary with `string.ascii_lowercase` as keys and a `list` with letters repeated based on their frequency as values.

`src.ch01.challenge.c1_foreign_bar_chart.`**`main`**()
Demonstrates the Foreign Bar Chart.

src.ch01.challenge.c2_name_generator module

Generate pseudo-random names from a list of names.

src.ch01.challenge.c2_name_generator.**add_name_to_key**(*name: str*, *dictionary: dict*, *key: str*) → None

Add name to key in dictionary.

Add **name** to **dictionary** under **key** if not already present.

Parameters

- **name** (*str*) – Name to add to **dictionary**.
- **key** (*str*) – Key to add **name** under.
- **dictionary** (*dict*) – Dictionary to add **name** to.

Returns None. **name** is added under **key** if not present, **dictionary** is unchanged otherwise.

Raises **TypeError** – If **name** and **key** aren't str or if **dictionary** isn't a dict.

src.ch01.challenge.c2_name_generator.**build_name_list**(*folderpath: str*) → list

Build name list from folder.

Builds list of names from name files in given folder.

Parameters **folderpath** (*str*) – Path to folder with name files.

Returns List with names from **folderpath**.

Raises **IndexError** – If **folderpath** has no .txt files.

src.ch01.challenge.c2_name_generator.**generate_name**(*name_dict: dict*) → str

Generate pseudo-random name.

Use names in dictionary to generate a random name.

Parameters **name_dict** – Dictionary from *split_names()*.

Returns String with a random name.

Raises **KeyError** – If there aren't three keys in the dictionary.

Note: Only add middle name between 1/3 and 1/4 of the time.

---

src.ch01.challenge.c2_name_generator.**main**()

Demonstrate name generator.

src.ch01.challenge.c2_name_generator.**name_generator**(*folderpath: str*) → str

Wrap generate_name, split_names, and build_name_list.

Passes given **folderpath** through *build_name_list()* to get the names in a list, then *split_names()* to split them into a dict, and finally through *generate_name()* to make the actual name.

Parameters **folderpath** (*str*) – Path to folder with name files.

Returns String with pseudo-random name.

src.ch01.challenge.c2_name_generator.**read_from_file**(*filepath: str*) → list

Read from file.

Reads lines from text file and returns a list.

Parameters **filepath** (*str*) – Path to file with names.

Returns List with each line from the file as an element.

Note: Removes trailing whitespaces.

---

src.ch01.challenge.c2_name_generator.**split_names**(*name_list: list*) → dict
Split names from list of names.

Splits first, middle, and last names from a given list of names.

Parameters **name_list** (*list*) – List with names as elements.

Returns Dictionary of lists with `first`, `middle`, and `last` as keys and names as values.

Raises

- **TypeError** – If given name list is not a `list` or `tuple`.
- **ValueError** – If given name list is empty.

Note: Drops suffix and adds nickname to middle names.

Module contents

Chapter 1 Challenge Projects.

src.ch01.challenge.**ADD_KEYS_ERROR**
String with `TypeError` for *add_keys_to_dict()*.

Type str

src.ch01.challenge.**SPLIT_NAME_LIST_ERROR**
String with `TypeError` for *split_names()*.

Type str

src.ch01.challenge.**SPLIT_NAME_EMPTY_ERROR**
Sting with `ValueError` for *split_names()*.

Type str

src.ch01.challenge.**ADD_NAME_TO_KEY_ERROR**
String with `TypeError` for *add_name_to_key()*.

Type str

src.ch01.challenge.**GENERATE_NAME_ERROR**
String with `KeyError` for *generate_name()*.

Type str

src.ch01.challenge.**BUILD_LIST_ERROR**
String with `IndexError` for *build_name_list()*.

Type str

src.ch01.practice package

Submodules

src.ch01.practice.p1_pig_latin module

Takes a word as input and returns its Pig Latin equivalent.

src.ch01.practice.p1_pig_latin.**encode**(*word: str*) → str
Check if word starts with vowel, then translate to Pig Latin.

If a word begins with a consonant, move the consonant to the end of the word and add 'ay' to the end of the new word. If a word begins with a vowel in *VOWELS*, add 'way' to the end of the word.

Parameters **word** (*str*) – Word to encode to Pig Latin.

Returns Encoded Pig Latin word.

Raises **TypeError** – If **word** is not a string.

src.ch01.practice.p1_pig_latin.**main**()
Demonstrate Pig Latin encoder.

src.ch01.practice.p2_poor_bar_chart module

Takes a sentence as input and returns a 'bar chart' of each letter.

src.ch01.practice.p2_poor_bar_chart.**freq_analysis**(*sentence: str*) → dict
Perform frequency analysis of letters in sentence.

Iterate through each letter in the sentence and add it to a dictionary of lists using `collections.defaultdict`.

Parameters **sentence** (*str*) – String to count letters of.

Returns `defaultdict` with each letter as keys and a `list` with letters repeated based on their frequency as values.

Example

```
>>> from src.ch01.practice.p2_poor_bar_chart import freq_analysis
>>> test = 'aaabbbccc'
>>> freq_analysis(test)
defaultdict(<class 'list'>, {'a': ['a', 'a', 'a'],
                             'b': ['b', 'b', 'b'],
                             'c': ['c', 'c', 'c']})
```

Raises **TypeError** – If **sentence** is not a string.

src.ch01.practice.p2_poor_bar_chart.**main**()
Demonstrates the Poor Bar Chart.

src.ch01.practice.p2_poor_bar_chart.**print_bar_chart**(*freq_dict: dict*) → None
Print dictionary to terminal.

Use `pprint.pprint()` to print dictionary with letter frequency analysis to terminal.

Parameters **freq_dict** (*dict*) – Dictionary with frequency analysis from *freq_analysis()*.

Returns None. Prints **freq_dict**.

Raises **TypeError** – If **freq_dict** is not a dictionary.

Module contents

Chapter 1 Practice Projects.

src.ch01.practice.**VOWELS**
Tuple containing characters of the English vowels (except for 'y')

Type tuple

src.ch01.practice.**ENCODE_ERROR**
String with TypeError for Pig Latin *encode()*.

Type str

src.ch01.practice.**FREQ_ANALYSIS_ERROR**
String with TypeError for Poor Bar Chart *freq_analysis()*.

Type str

src.ch01.practice.**PRINT_BAR_CHART_ERROR**
String with TypeError for Poor Bar Chart *print_bar_chart()*.

Type str

Module contents

Chapter 1.

src.ch02 package

Submodules

src.ch02.c1_recursive_palindrome module

Recursively determine if a word is a palindrome.

src.ch02.c1_recursive_palindrome.**main**(*word: str = None*) → None
Demonstrate the recursive palindrome tester.

This is only supposed to be a demo, but coverage necessitates excessiveness.

Parameters **word** (*str*) – Word to test if it is a palindrome.

Returns None. Identifies **word** as a palindrome.

src.ch02.c1_recursive_palindrome.**recursive_ispalindrome**(*word: str*) → bool
Recursively check if a word is a palindrome.

Parameters **word** (*str*) – String to check palindrome-ness.

Returns True if the word is a palindrome, False otherwise.

Raises **TypeError** – If **word** is not a string.

src.ch02.p1_cleanup_dictionary module

Cleanup word dictionary.

Various functions for cleaning up a word dictionary.

src.ch02.p1_cleanup_dictionary.**APPROVED_WORDS**
Words that should always appear in a word dictionary.

Type list

src.ch02.p1_cleanup_dictionary.**cleanup_dict**(*filepath: str*) → list
Wrap read_from_file and cleanup_list.

Passes given **filepath** through *read_from_file()* to get a list of words, then *cleanup_list()* to remove single letter words.

Parameters **filepath** (*str*) – String with path to word dictionary file.

Returns List with words as elements excluding single letter words.

src.ch02.p1_cleanup_dictionary.**cleanup_list**(*word_list: list*) → list
Cleanup word list.

Remove single letter words from a list of words.

Parameters **word_list** (*list*) – List with words as elements.

Returns List with words as elements excluding single letter words.

Raises **IndexError** – If **word_list** is empty.

src.ch02.p1_cleanup_dictionary.**cleanup_list_more**(*word_list: list*) → list
Cleanup word list even more.

First, remove words with apostrophes, double letter words, duplicates, and words with letters not in string.ascii_lowercase from a list of words. Then, add *APPROVED_WORDS* back into list. Finally, sort list.

Parameters **word_list** (*list*) – List with words as elements.

Returns Sorted list with words as elements excluding cleaned words and *APPROVED_WORDS* added.

Raises **IndexError** – If **word_list** is empty.

src.ch02.p1_cleanup_dictionary.**main**()
Demonstrate cleanup dictionary.

Module contents

Chapter 2.

src.ch02.**DICTIONARY_FILE_PATH**
String with path to Ubuntu 18.04.2's American English dictionary file.

Type str

src.ch02.**CLEANUP_LIST_ERROR**
String with IndexError for Cleanup Dictionary *cleanup_list()*.

Type str

src.ch02.**RECURSIVE_ISPALINDROME_ERROR**
String with TypeError for Recursive Palindrome *recursive_ispalindrome()*.

Type str

src.ch03 package

Submodules

src.ch03.c1_anagram_generator module

Generate phrase anagrams from a word or phrase.

src.ch03.c1_anagram_generator.**anagram_generator**(*word: str*) → list
Generate phrase anagrams.

Make phrase anagrams from a given word or phrase.

Parameters **word** (*str*) – Word to get phrase anagrams of.

Returns list of phrase anagrams of **word**.

src.ch03.c1_anagram_generator.**extend_anagram_dict**(*word_list: list*, *dictionary: dict*)
Extend an anagram dictionary.

Adds words from given word list to a given anagram dictionary.

Parameters

- **word_list** (*list*) – List of words to add to anagram dictionary.
- **dictionary** (*dict*) – Anagram dictionary to add words to.

Returns None. If words in **word_list** are in **dictionary** they are not added. Otherwise, they are added.

src.ch03.c1_anagram_generator.**find_anagram_phrases**(*phrases: list*, *word: str*, *anagram_dict: dict*, *phrase: list*) → None
Find anagram phrases.

Recursively finds anagram phrases of **word** by removing unusable words from the **anagram_dict**, finding remaining anagrams given the **phrase**, then adding any found anagram phrases to **phrases**.

Parameters

- **phrases** (*list*) – List of anagram phrases.
- **word** (*str*) – Current word to find anagram phrases of.
- **anagram_dict** (*dict*) – Current anagram dictionary to find anagrams with.
- **phrase** (*list*) – Current anagram phrase candidate.

Returns None. **phrases** is updated with any found anagram phrases.

src.ch03.c1_anagram_generator.**find_anagrams**(*word: str*, *anagram_dict: dict*) → list
Find anagrams in word.

Find all anagrams in a given word (or phrase) using anagram dictionary.

Parameters

- **word** (*str*) – Word to find anagrams of.

- **anagram_dict** – Dictionary from *get_anagram_dict()*.

Returns list of str with all anagrams in **word**.

src.ch03.c1_anagram_generator.**get_anagram_dict**(*word_list: list*) → dict
Get an anagram dictionary from word_list.

Get the ID of each word in **word list** and add it to a dictionary with the ID as the key.

Parameters **word_list** (*list*) – List of words to make into anagram dictionary.

Returns defaultdict of list with an ID (int) as the key and words whose product of letters equal that ID as values.

src.ch03.c1_anagram_generator.**get_id**(*word: str*) → int
Get ID number of word.

Assign a unique prime number to each letter in ascii_lowercase. The product of each letter in **word** is its ID number.

Parameters **word** (*str*) – Word to get ID of.

Returns int representing ID of **word**.

src.ch03.c1_anagram_generator.**get_primes**(*length: int = 26*, *min_prime: int = 2*, *max_prime: int = 101*) → list
Get list of primes.

Given a length, minimum, and maximum prime number, return a list of prime numbers.

Parameters

- **length** (*int*) – Number of prime numbers to return. Defaults to 26.
- **min_prime** (*int*) – Smallest prime number to return. Defaults to 2.
- **max_prime** (*int*) – Largest prime number to return. Defaults to 101.

Returns list of **length** prime numbers with **min_prime** as the smallest prime number and **max_prime** as the largest prime number in the list.

src.ch03.c1_anagram_generator.**main**()
Demonstrate the Anagram Generator.

src.ch03.c1_anagram_generator.**multi_get_anagram_dict**(*word_list: list*) → dict
Multithreaded get anagram dictionary.

Uses os.cpu_count() and threading.Thread to use all CPUs to make an anagram dictionary with the intent of being more efficient than *get_anagram_dict()*.

Parameters **word_list** (*list*) – List of words to make into anagram dictionary.

Returns defaultdict of list with an ID (int) as the key and words whose product of letters equal that ID as values.

> **Warning:** Avoids race conditions by heavily relying on CPython's Global Interpreter Lock. More info about Thread Objects.

src.ch03.c1_anagram_generator.**remove_unusable_words**(*anagram_dict: dict*, *usable_letters: list*) → dict
Remove unusable words from anagram dictionary.

Creates new anagram dictionary by including only IDs that can be IN **usable_letters**.

Parameters

- **anagram_dict** (*dict*) – Anagram dictionary to prune.
- **usable_letters** (*list*) – List of letters that must be used.

Returns `defaultdict` of `list` with an ID (`int`) as the key and words whose product of letters equal that ID as values.

`src.ch03.c1_anagram_generator.`**split**(*a_list: list*, *parts: int*) → list

Split a list into parts.

Split given list into given number of parts.

Parameters

- **a_list** (*list*) – List to split.
- **parts** (*int*) – Number of parts to split list into.

Returns List of lists with **a_list** split into **parts**.

Example

```
>>> import src.ch03.c1_anagram_generator.split as split
>>> some_list = ['this', 'is', 'a', 'list']
>>> split_list = split(some_list, 2)
>>> print(split_list)
[['this', 'is'], ['a', 'list']]
```

src.ch03.p1_digram_counter module

Counts the occurrence of all possible digrams of a word in a dictionary.

`src.ch03.p1_digram_counter.`**count_digrams**(*digrams: set*, *dict_list: list*) → dict

Count digrams in word dictionary.

Count frequency of each digram in the set in a word dictionary list.

Parameters

- **digrams** (*set*) – Set of digrams to count frequency of.
- **dict_list** (*list*) – Word dictionary list.

Returns `Counter` with digrams as keys and their counts as values.

Raises **TypeError** – If **digrams** isn't a set or if **dict_list** isn't a list.

`src.ch03.p1_digram_counter.`**digram_counter**(*word: str*, *dict_file: str = '/usr/share/dict/american-english'*) → dict

Wrap get_digrams, count_digrams, and read_from_file.

Send **word** through *get_digrams()* to get a set of digrams which is then passed through *count_digrams()* along with the list made by passing **dict_file** through *read_from_file()*.

Parameters

- **word** (*str*) – Word to get digrams of.
- **dict_file** (*str*) – Path of dictionary file to get a frequency analysis of each digram. Defaults to *DICTIONARY_FILE_PATH*.

Returns `Counter` with digrams as keys and their counts as values.

src.ch03.p1_digram_counter.**get_digrams**(*word: str*) → set
Get a set of digrams given a word.

Generate all possible digrams of a given word.

Parameters **word** (*str*) – String to get digrams of.

Returns set of all possible digrams of the given word.

Raises **TypeError** – If **word** isn't a string.

src.ch03.p1_digram_counter.**main**()
Demonstrate the digram counter.

Module contents

Chapter 3.

src.ch03.**GET_DIGRAMS_ERROR**
String with TypeError for *get_digrams()*.

Type str

src.ch03.**COUNT_DIGRAMS_ERROR**
String with TypeError for *count_digrams()*.

Type str

src.ch04 package

Subpackages

src.ch04.challenge package

Submodules

src.ch04.challenge.c1_encode_route module

Encode a route cipher and replace code words.

src.ch04.challenge.c1_encode_route.**encode_route**(*plaintext: str*, *keys: list*, *rows: int*) → list
Encode plaintext message with route cipher.

Clean **plaintext** with *format_plaintext()*, replace sensitive intel with *replace_words()*, fill with dummy words using *fill_dummy()* until **keys** and **rows** are factors, then encrypt with a route cipher using **keys**.

Parameters

- **plaintext** (*str*) – Plaintext message to encode with route cipher.
- **keys** (*list*) – List of positive/negative integers representing cipher route.
- **rows** (*int*) – Number of rows to use in the route cipher table.

Returns List of strings of transposed words.

Note: Assumes vertical encoding routes.

src.ch04.challenge.c1_encode_route.**fill_dummy**(*plainlist: list*, *factors: list*, *dummy_words: list = None*) → list

Fill a plainlist with dummy words.

Adds pseudorandom dummy words to the end until the factors of the length of **plainlist** includes **factors**.

Parameters

- **plainlist** (*list*) – List of words of plaintext message.
- **factors** (*list*) – List of integers that must be factors of the length of **plainlist**.
- **dummy_words** (*list*) – List of dummy words to use as filler. If not provided, defaults to *DICTIONARY_FILE_PATH* using *cleanup_dict()*.

Returns Same list as **plainlist**, but with dummy words added.

src.ch04.challenge.c1_encode_route.**format_plaintext**(*plaintext: str*) → list

Format plaintext message for encoding.

Prepare **plaintext** for route cipher encoding. Convert to lowercase, remove punctuation.

Parameters **plaintext** (*str*) – Plaintext message to format.

Returns List of strings of each word in plaintext message.

src.ch04.challenge.c1_encode_route.**main**()

Demonstrate the route cipher encoder.

src.ch04.challenge.c1_encode_route.**replace_words**(*plainlist: list*, *code_words: dict = None*) → list

Replace sensitive words with code words.

Replace words that shouldn't be transmitted with code words.

Parameters

- **plainlist** (*list*) – List of strings of each word in plaintext message.
- **code_words** (*dict*) – Dictionary of sensitive words and their code words. If not provided, defaults to the book's code words. Use lowercase strings in dictionary.

Returns Same list, but with sensitive words replaced with code words.

src.ch04.challenge.c2_encode_rail module

Encode message with a 3-rail fence cipher.

src.ch04.challenge.c2_encode_rail.**encode_rail**(*plaintext: str*, *split: int = 5*) → str

Encode rail fence cipher.

Encode **plaintext** with a 3-rail fence cipher. Scrub the plaintext with *format_plaintext()*, then encrypt it with *split_rails()*.

Parameters

- **plaintext** (*str*) – Message to encrypt with 3-rail fence cipher.
- **split** (*int*) – How many letter segments to split message into. Defaults to 5.

Returns String with encrypted message split into **split** chunks for easier transmission.

src.ch04.challenge.c2_encode_rail.**main**()
Demonstrate 3-rail fence cipher encoder.

src.ch04.challenge.c2_encode_rail.**split_rails**(*plaintext: str*) → str
Split plaintext into 3 rails for encryption.

Split the rails where the top rail is every 4th letter, the middle rail is every other letter starting at 1, and the bottom rail is every 4th letter starting at 2. After splitting, concatenate each rail and return the result.

Parameters **plaintext** (*str*) – Plain text message without spaces or punctuation.

Returns String with message encrypted using 3 rail fence cipher.

Module contents

Chapter 4 Challenge Projects.

src.ch04.practice package

Submodules

src.ch04.practice.p1_hack_lincoln module

Hack route cipher sent by Abraham Lincoln.

src.ch04.practice.p1_hack_lincoln.**decode_route**(*keys: list*, *cipherlist: list*) → list
Decode route cipher.

Decode **cipherlist** encoded with a route cipher using **keys**.

Parameters

- **keys** (*list*) – List of signed, integer keys.
- **cipherlist** (*list*) – List of strings representing encoded message.

Returns List of strings representing plaintext message.

Note: Assumes vertical encoding route.

---

src.ch04.practice.p1_hack_lincoln.**get_factors**(*integer: int*) → list
Get factors of integer.

Calculate factors of a given integer.

Parameters **integer** (*int*) – Number to get factors of.

Returns List of integer factors of **integer**.

src.ch04.practice.p1_hack_lincoln.**hack_route**(*ciphertext: str*) → None
Hack route cipher.

Hack route cipher by using *get_factors()* to find all possible key lengths. Then use *keygen()* to generate all possible keys and pass each one through *decode_route()*.

Parameters **ciphertext** (*str*) – Message encoded with route cipher.

Returns None. Prints all possible decoded messages.

src.ch04.practice.p1_hack_lincoln.**keygen**(*length: int*) → list
Generate all possible route cipher keys.

Generates a list of all possible route cipher keys of **length**.

Parameters **length** (*int*) – Length of route cipher key.

Returns List of lists of integers representing all possible route cipher keys of **length**.

Example

```
>>> from src.ch04.practice.p1_hack_lincoln import keygen
>>> keygen(2)
[[-1, -2], [-1, 2], [1, -2], [1, 2]]
```

src.ch04.practice.p1_hack_lincoln.**main**()
Demonstrate hack of Lincoln's route cipher.

src.ch04.practice.p2_identify_cipher module

Identify letter transposition or substitution cipher.

src.ch04.practice.p2_identify_cipher.**identify_cipher**(*ciphertext: str*, *threshold: float*) → bool
Identify letter transposition or substitution cipher.

Compare most frequent letters in **ciphertext** with the most frequent letters in the English alphabet. If above **threshold**, it is a letter transposition cipher. If not, it is a letter substitution cipher.

Parameters

- **ciphertext** (*str*) – Encrypted message to identify.
- **threshold** (*float*) – Percent match in decimal form.

Returns `True` if the **ciphertext** is a letter transposition cipher. `False` otherwise.

src.ch04.practice.p2_identify_cipher.**is_substitution**(*ciphertext: str*) → bool
Identify letter substitution cipher.

Wrapper for *identify_cipher()*. **threshold** defaults to `0.45`.

Parameters **ciphertext** (*str*) – Encrypted message to identify.

Returns `True` if the **ciphertext** is a letter substitution cipher. `False` otherwise.

src.ch04.practice.p2_identify_cipher.**is_transposition**(*ciphertext: str*) → bool
Identify letter transposition cipher.

Wrapper for *identify_cipher()*. **threshold** defaults to `0.75`.

Parameters **ciphertext** (*str*) – Encrypted message to identify.

Returns `True` if the **ciphertext** is a letter transposition cipher. `False` otherwise.

src.ch04.practice.p2_identify_cipher.**main**(*ciphertext: str = None*) → None
Demonstrate the cipher identifier.

This is only supposed to be a demo, but coverage necessitates excessiveness.

Parameters **ciphertext** (*str*) – Encrypted letter transposition or letter substitution cipher to demonstrate.

Returns None. Identifies **ciphertext**'s cipher.

src.ch04.practice.p2_identify_cipher_deco module

Identify letter transposition or substitution cipher using decorator.

Note: **Not** part of the book, I was just curious about decorators and decided to tinker with them a bit.

src.ch04.practice.p2_identify_cipher_deco.**identify**(*threshold: float = 0.5*)
Make decorator for identify_cipher.

Decorator factory to replace a decorated function with *identify_cipher()*. A bit like going around the world to reach the teleporter across the street, but at import time instead of runtime, so it doesn't matter.

Luciano Ramalho's book *Fluent Python* appropriately calls decorators "syntactic sugar" when they aren't used in classes. It also references the wrapt module's blog on GitHub for a deeper explanation of decorators.

Not sure what a decorator factory would be called... syntactic caramel?

Parameters **threshold** (*float*) – Percent match in decimal form.

Returns Whatever the output of identify_cipher() would be given the decorated function's input.

src.ch04.practice.p2_identify_cipher_deco.**is_substitution**(*ciphertext: str*) → bool
Identify letter substitution cipher.

Empty function to wrap with *identify_cipher()* using *identify()*. **threshold** defaults to 0.45.

Parameters **ciphertext** (*str*) – Encrypted message to identify.

Returns True if the **ciphertext** is a letter substitution cipher. False otherwise.

src.ch04.practice.p2_identify_cipher_deco.**is_transposition**(*ciphertext: str*) → bool
Identify letter transposition cipher.

Empty function to wrap with *identify_cipher()* using *identify()*. **threshold** defaults to 0.75.

Parameters **ciphertext** (*str*) – Encrypted message to identify.

Returns True if the **ciphertext** is a letter transposition cipher. False otherwise.

src.ch04.practice.p3_get_keys module

Get route cipher key from user and store as dictionary.

Note: Assumes vertical cipher routes.

src.ch04.practice.p3_get_keys.**get_keys**() → list
Get route cipher keys from user.

User only has to enter positive/negative integers. Each gets added to a list and returned when the user has no other keys to add.

Returns List of integers as column numbers and positive/negative values as route direction.

src.ch04.practice.p3_get_keys.**key_to_dict**(*keys: list*) → dict
Convert route cipher key to dictionary.

Take a route cipher key in list format where integers are column numbers and positive/negative is the route direction and convert to a dictionary where the column numbers are keys and the route direction as up/down are the values.

Parameters **keys** (*list*) – List of integers with direction as positive/negative.

Returns Integers keys and up/down as values.

src.ch04.practice.p3_get_keys.**main**()
Demonstrate getting route cipher keys from the user.

src.ch04.practice.p4_generate_keys module

Generate route cipher keys for brute-forcing a route cipher.

Already implemented with *keygen()*, but this version will return a list of tuples.

src.ch04.practice.p4_generate_keys.**generate_keys**(*length: int*) → list
Generate all possible route cipher keys.

Generates a list of all possible route cipher keys of **length**.

Parameters **length** (*int*) – Length of route cipher key.

Returns List of tuples of integers representing all possible route cipher keys of **length**.

src.ch04.practice.p4_generate_keys.**main**()
Demonstrate the key generator.

src.ch04.practice.p5_hack_route module

Another way to hack a route cipher.

Already implemented in *p1_hack_lincoln*, but this version will use the building blocks made in *p2_identify_cipher*, *p3_get_keys*, and *p4_generate_keys*.

src.ch04.practice.p5_hack_route.**decode_route**(*keys: dict*, *cipherlist: list*) → list
Decode route cipher.

Decode **cipherlist** encoded with a route cipher using **keys**.

Parameters

- **keys** (*dict*) – up/down dictionary with column numbers as keys.
- **cipherlist** (*list*) – List of strings representing encoded message.

Returns List of strings representing plaintext message.

Note: Assumes vertical encoding route.

---

src.ch04.practice.p5_hack_route.**hack_route**(*ciphertext: str*, *columns: int*) → None
Hack route cipher using brute-force attack.

Determine if **ciphertext** is a transposition cipher. If so, use **columns** to generate all possible keys. Convert each key to an up/down dictionary for each route to take, then print the result of each key.

Parameters

- **ciphertext** (*str*) – Route cipher encoded string to hack.
- **columns** (*int*) – Number route cipher columns.

Returns None. Prints all possible decoded messages.

src.ch04.practice.p5_hack_route.**main**()
Demonstrate the route cipher hacker.

Module contents

Chapter 4 Practice Projects.

Module contents

Chapter 4.

1.1.2 Module contents

impracticalpythonprojects.

Example implementations of the projects in Impractical Python Projects.

MIT License

Jose A. Lerma III

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

S

INDEX

H

I

K

M

N

P

R

S

V