

Kinoshita Daisuke

<https://www.instagram.com/daisuke23888/>

<https://s3b.astro.ncu.edu.tw/ipype/>

08 May 2025

基礎 Python 程式設計練習

— Session 10 —

<https://meet.google.com/wpx-yabb-tka>

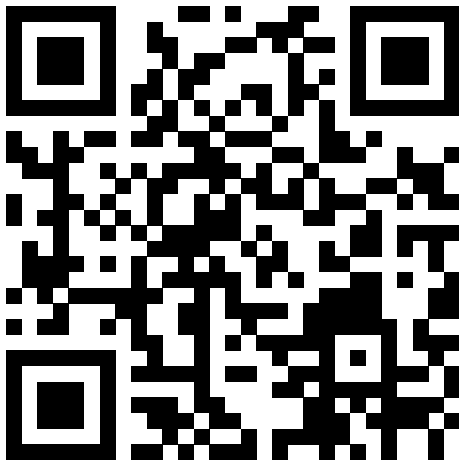
進來的人請跟我說一聲

有人進來我們就開始

**Please talk to me, when you
come in.**

**When you come in, we start the
session.**

- Date/Time: 08 May 2025
- Google Meet: <https://meet.google.com/wpx-yabb-tka>
- Language: English, or Chinese, or Japanese
- Anyone can join this activity.
- This activity is organised in conjunction with the course “Doing Astrophysics using Python”. If you take the course “Doing Astrophysics using Python” and you would like to study basic Python programming, you are highly encouraged to join this activity.
- Activity web page: <https://s3b.astro.ncu.edu.tw/ipype/>



<https://s3b.astro.ncu.edu.tw/ipyte/>

Some notes for you

- Ask a question to me at any time, if you have any difficulty.
- When you finish writing your code, tell me about it.
- Show me your source code, if you have an error for your code and cannot find a way to fix it.
- Tell me if you want me to provide a sample code for you.
- Be active, then you gain more.

- 如果想要問問題，請跟我說。什麼時候都可以。
- 你的程式寫完的時候，請跟我說一聲。
- 如果你不確定你自己寫的程式寫得好不好，請給我看你寫的原始碼。我跟你說我的意見。
- 如果你想要看我的寫法，請跟我說。我寫給你看。
- 比較主動一點，你的收穫比較多。

Downloading the data file

- Visit the following web page, and download the data file and ReadMe file.
 - “Milky Way molecular clouds from 12CO”
 - <https://cdsarc.cds.unistra.fr/viz-bin/cat/J/ApJ/834/57>

Downloading the data file

Navigation bar: CDS, PORTAL, SIMBAD, VizieR, ALADIN, XMATCH, OTHERS, HELP ?

Milky Way molecular clouds from ^{12}CO : J/ApJ/834/57

Access to: [VizieR](#) [FTP](#) [ReadMe](#) [TAP](#) [Xmatch](#) [Download notebook](#)

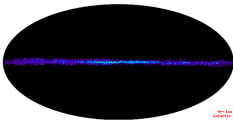
Authors : Milville-Deschenes M.-A. , Murray N., Lee E.J.

VizieR DOI : [10.26093/cds/vizieR.18340057](https://cdsarc.cds.unistra.fr/viz-bin/doi/10.26093/cds/vizieR.18340057) [Cite](#)

Bibcode : [2017ApJ...834...57M \(ADS\)](#)

UAT : Molecular clouds, CO line emission, Galaxy planes, Milky Way Galaxy, Interstellar medium

Observation (OC)



Inserted into VizieR : 12-Jun-2017
Last modification : 05-Jun-2019

Article Origin: [Description](#) [See also](#) [History](#) [Prov](#) [FTP](#) [VizieR](#)

Physical properties of molecular clouds for the entire Milky Way disk. (2017)
[Go to the original article \(10.3847/1538-4357/834/1/57\)](#)

Keywords : Galaxy general; ISM: clouds; ISM: general; ISM: kinematics and dynamics; methods: data analysis; turbulence

Abstract: This study presents a catalog of 8107 molecular clouds that covers the entire Galactic plane and includes 98% of the ^{12}CO emission observed within $b \pm 5^\circ$. The catalog was produced using a hierarchical cluster identification method applied to the result of a Gaussian decomposition of the Dame+ (2001ApJ...547..792D) data. The total H_2 mass in the catalog is $1.2 \times 10^9 M_\odot$, in agreement with previous estimates. We find that 30% of the sight lines intersect only a single cloud, with another 25% intersecting only two clouds. The most probable cloud size is $R \sim 30\text{pc}$. We find that $M(\text{propt})R^{2.2} \pm 0.2^*$, with no correlation between the cloud surface density, (Sigma), and R. In contrast with the general idea, we find a rather large range of ...[\(more\)](#)

[American Astronomical Society policies](#)

<https://cdsarc.cds.unistra.fr/viz-bin/cat/J/ApJ/834/57>

- Make your own Python script to open the data file “table1.dat.gz”, extract galactic longitude (GLON), galactic latitude (GLAT), distance flag (INF), near kinematic distance (Dnear), and far kinematic distance (Dfar) for each molecular cloud in the catalogue.

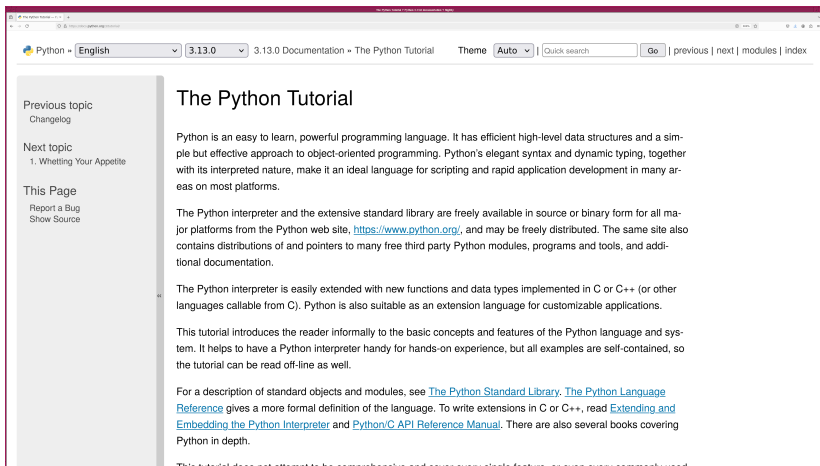
- Make your own Python script to open the data file "table1.dat.gz", extract galactic longitude (GLON), galactic latitude (GLAT), distance flag (INF), near kinematic distance (Dnear), and far kinematic distance (Dfar) for each molecular cloud in the catalogue, check the distance flag and find more likely distance (Dnear or Dfar).

- Make your own Python script to open the data file “table1.dat.gz”, extract galactic longitude (GLON), galactic latitude (GLAT), distance flag (INF), near kinematic distance (Dnear), and far kinematic distance (Dfar) for each molecular cloud in the catalogue, check the distance flag and find more likely distance (Dnear or Dfar), calculate galactocentric coordinate (x_{gal} , y_{gal} , z_{gal}).
 - Assume 8.34 kpc for the distance of the Sun from the Galactic centre.
 - Assume 25 pc for the distance of the Sun from the Galactic mid-plane.
 - For the calculation of galactocentric coordinates, read the Section 2.4.1 “Physical Properties” of the paper “A UNIFORM CATALOG OF MOLECULAR CLOUDS IN THE MILKY WAY”.
 - <https://iopscience.iop.org/article/10.3847/0004-637X/822/1/52>

- Make your own Python script to open the data file “table1.dat.gz”, extract galactic longitude (GLON), galactic latitude (GLAT), distance flag (INF), near kinematic distance (Dnear), and far kinematic distance (Dfar) for each molecular cloud in the catalogue, check the distance flag and find more likely distance (Dnear or Dfar), calculate galactocentric coordinate $(x_{gal}, y_{gal}, z_{gal})$, and then visualise locations of molecular clouds on (x_{gal}, y_{gal}) plane using Matplotlib.
 - Assume 8.34 kpc for the distance of the Sun from the Galactic centre.
 - Assume 25 pc for the distance of the Sun from the Galactic mid-plane.
 - For the calculation of galactocentric coordinates, read the Section 2.4.1 “Physical Properties” of the paper “A UNIFORM CATALOG OF MOLECULAR CLOUDS IN THE MILKY WAY”.
 - <https://iopscience.iop.org/article/10.3847/0004-637X/822/1/52>

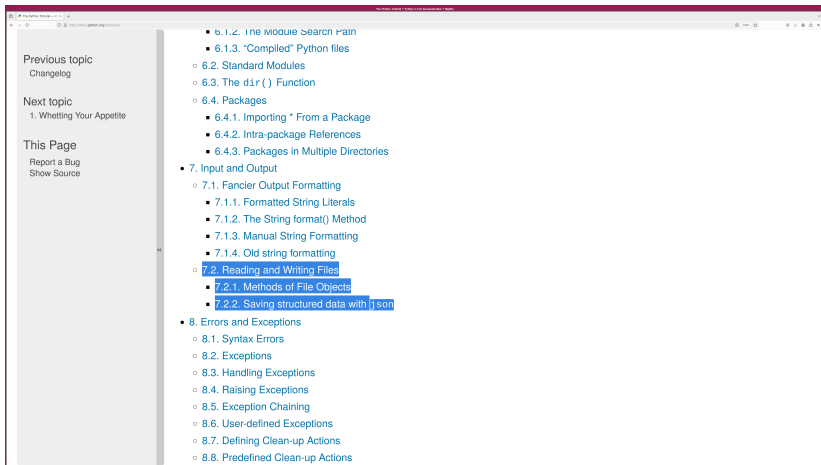
- We will have next meeting on 15 May 2025 .
 - Google Meet: <https://meet.google.com/wpx-yabb-tka>
- We start the activity at 20:00.

"The Python Tutorial"



<https://docs.python.org/3/tutorial/>

"The Python Tutorial"



<https://docs.python.org/3/tutorial/>

Table of Contents

- 7. Input and Output
 - 7.1. Fancier Output Formatting
 - 7.1.1. Formatted String Literals
 - 7.1.2. The String format() Method
 - 7.1.3. Manual String Formatting
 - 7.1.4. Old string formatting
 - 7.2. Reading and Writing Files
 - 7.2.1. Methods of File Objects
 - 7.2.2. Saving structured data with json

Previous topic

6. Modules

Next topic

8. Errors and Exceptions

This Page

[Report a Bug](#)
[Show Source](#)

7.2. Reading and Writing Files

`open()` returns a [file object](#), and is most commonly used with two positional arguments and one keyword argument: `open(filename, mode, encoding=None)`

```
>>> f = open('workfile', 'w', encoding="utf-8")>>>
```

The first argument is a string containing the filename. The second argument is another string containing a few characters describing the way in which the file will be used. *mode* can be `'r'` when the file will only be read, `'w'` for only writing (an existing file with the same name will be erased), and `'a'` opens the file for appending; any data written to the file is automatically added to the end. `'r+'` opens the file for both reading and writing. The *mode* argument is optional; `'r'` will be assumed if it's omitted.

Normally, files are opened in *text mode*, that means, you read and write strings from and to the file, which are encoded in a specific *encoding*. If *encoding* is not specified, the default is platform dependent (see [open\(\)](#)). Because UTF-8 is the modern de-facto standard, `encoding="utf-8"` is recommended unless you know that you need to use a different encoding. Appending a `'b'` to the mode opens the file in *binary mode*. Binary mode data is read and written as [bytes](#) objects. You can not specify *encoding* when opening file in binary mode.

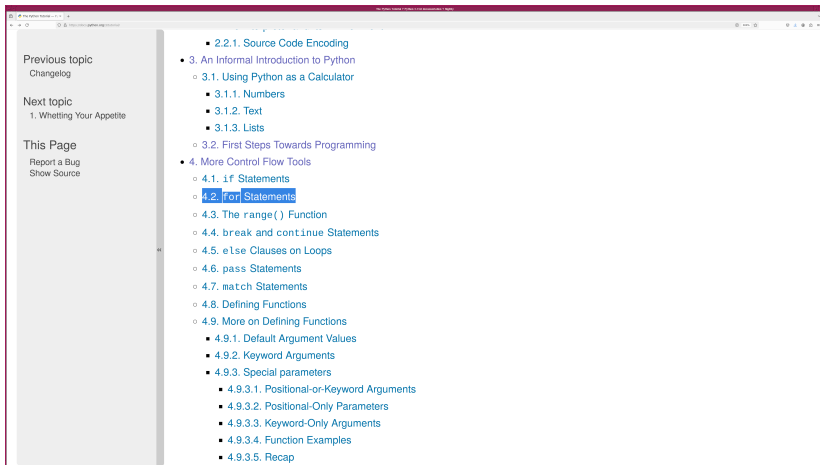
In text mode, the default when reading is to convert platform-specific line endings (`\n` on Unix, `\r\n` on Windows) to just `\n`. When writing in text mode, the default is to convert occurrences of `\n` back to platform-specific line endings. This behind-the-scenes modification to file data is fine for text files, but will corrupt binary data like that in JPEG or EXE files. Be very careful to use binary mode when reading and writing such files.

It is good practice to use the [with](#) keyword when dealing with file objects. The advantage is that the file is properly closed after its suite finishes, even if an exception is raised at some point. Using `with` is also much shorter

<https://docs.python.org/3/tutorial/inputoutput.html#reading-and-writing-files>

- To learn about the “for” statement, read the section 4.2 “for Statements”.
 - <https://docs.python.org/3/tutorial/controlflow.html#for-statements>

"The Python Tutorial"



<https://docs.python.org/3/tutorial/>

Table of Contents

- 4. More Control Flow Tools
 - 4.1. `if` Statements
 - 4.2. `for` Statements
 - 4.3. The `range()` Function
 - 4.4. `break` and `continue` Statements
 - 4.5. `else` Clauses on Loops
 - 4.6. `pass` Statements
 - 4.7. `match` Statements
 - 4.8. Defining Functions
 - 4.9. More on Defining Functions
 - 4.9.1. Default Argument Values
 - 4.9.2. Keyword Arguments
 - 4.9.3. Special parameters
 - 4.9.3.1. Positional-or-Keyword Arguments
 - 4.9.3.2. Positional-Only Parameters
 - 4.9.3.3. Keyword-Only Arguments
 - 4.9.3.4. Function Examples
 - 4.9.3.5. Recap

4.2. for Statements

The `for` statement in Python differs a bit from what you may be used to in C or Pascal. Rather than always iterating over an arithmetic progression of numbers (like in Pascal), or giving the user the ability to define both the iteration step and halting condition (as C), Python's `for` statement iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence. For example (no pun intended):

```
>>> # Measure some strings:
>>> words = ['cat', 'window', 'defenestrate']
>>> for w in words:
...     print(w, len(w))
...
cat 3
window 6
defenestrate 12
```

Code that modifies a collection while iterating over that same collection can be tricky to get right. Instead, it is usually more straight-forward to loop over a copy of the collection or to create a new collection:

```
# Create a sample collection
users = {'Hans': 'active', 'Éléonore': 'inactive', '景太郎': 'active'}

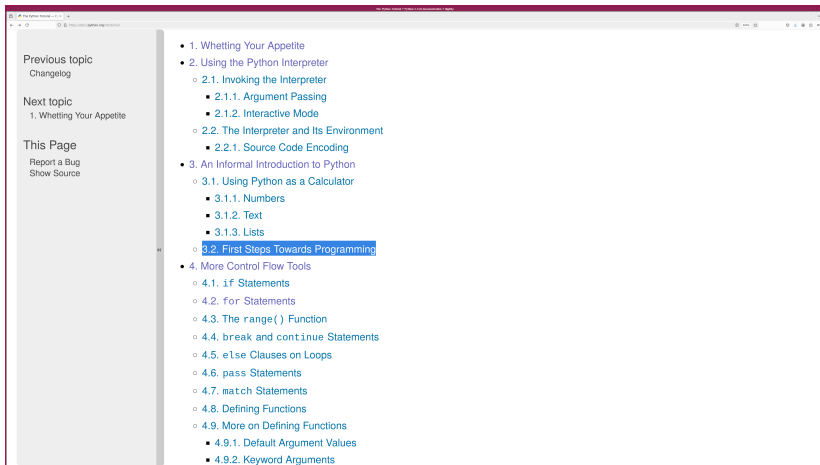
# Strategy: Iterate over a copy
for user, status in users.copy().items():
    if status == 'inactive':
        del users[user]

# Strategy: Create a new collection
active_users = {}
for user, status in users.items():
    if status == 'active':
        active_users[user] = status
```

[https://docs.python.org/3/tutorial/controlflow.html#
for-statements](https://docs.python.org/3/tutorial/controlflow.html#for-statements)

- To learn about the “while” statement, read the section 3.2 “First Steps Towards Programming”.
 - <https://docs.python.org/3/tutorial/introduction.html#first-steps-towards-programming>

"The Python Tutorial"



<https://docs.python.org/3/tutorial/>

<https://www.instagram.com/daisuke23888/>

The screenshot shows the Python 3.6.0 documentation page for '3.2. First Steps Towards Programming'. The left sidebar contains a 'Table of Contents' with links to '3. An Informal Introduction to Python' and its sub-sections, 'Previous topic', 'Next topic', and 'This Page'. The main content area has the title '3.2. First Steps Towards Programming' followed by a paragraph explaining that Python can be used for more complex tasks than simple arithmetic, using the 'Fibonacci series' as an example. Below this is a code block showing a Python script that calculates the first 10 Fibonacci numbers. The output of the script is displayed below the code. A paragraph follows, stating 'This example introduces several new features.' and then lists two key features: multiple assignment and the while loop.

Table of Contents

- 3. An Informal Introduction to Python
 - 3.1. Using Python as a Calculator
 - 3.1.1. Numbers
 - 3.1.2. Text
 - 3.1.3. Lists
 - 3.2. First Steps Towards Programming

Previous topic

- 2. Using the Python Interpreter

Next topic

- 4. More Control Flow Tools

This Page

- Report a Bug
- Show Source

3.2. First Steps Towards Programming

Of course, we can use Python for more complicated tasks than adding two and two together. For instance, we can write an initial sub-sequence of the [Fibonacci series](#) as follows:

```
>>> # Fibonacci series:
>>> # the sum of two elements defines the next
>>> a, b = 0, 1
>>> while a < 10:
...     print(a)
...     a, b = b, a+b
...
0
1
1
2
3
5
8
```

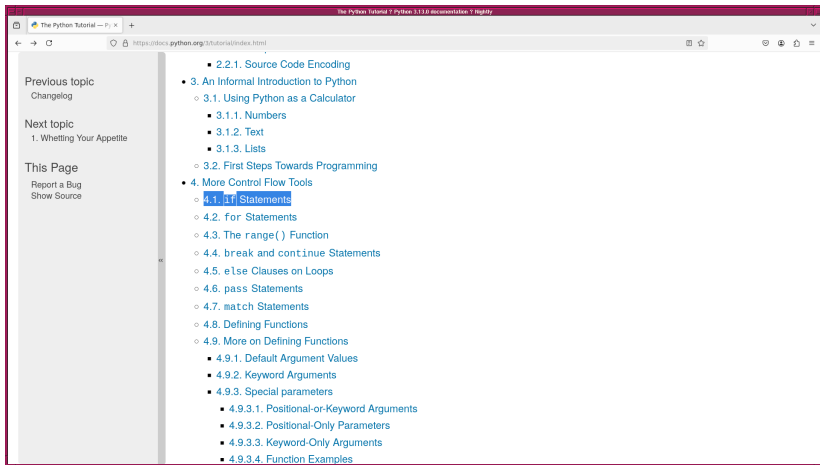
This example introduces several new features.

- The first line contains a *multiple assignment*: the variables `a` and `b` simultaneously get the new values 0 and 1. On the last line this is used again, demonstrating that the expressions on the right-hand side are all evaluated first before any of the assignments take place. The right-hand side expressions are evaluated from the left to the right.
- The `while` loop executes as long as the condition (here: `a < 10`) remains true. In Python, like in C, any non-zero integer value is true; zero is false. The condition may also be a string or list value, in fact any sequence; anything with a non-zero length is true, empty sequences are false. The test used in the example is a simple comparison. The standard comparison operators are written the same as in C: `<` (less than), `<=` (less than or equal to), `>` (greater than), `>=` (greater than or equal to), `==` (equal to), `!=` (not equal to), `in` (membership), `not in` (non-membership).

<https://docs.python.org/3/tutorial/introduction.html#first-steps-towards-programming>

- To learn about the "if" statement, read the section 4.1 "if Statements".
 - [https://docs.python.org/3/tutorial/controlflow.html#
if-statements](https://docs.python.org/3/tutorial/controlflow.html#if-statements)

"The Python Tutorial"



<https://docs.python.org/3/tutorial/>

"The Python Tutorial"

https://www.instagram.com/daisuke23888/

Table of Contents

- 4. More Control Flow Tools
 - 4.1. **if** Statements
 - 4.2. **for** Statements
 - 4.3. The **range()** Function
 - 4.4. **break** and **continue** Statements
 - 4.5. **else** Clauses on Loops
 - 4.6. **pass** Statements
 - 4.7. **match** Statements
 - 4.8. Defining Functions
 - 4.9. More on Defining Functions
 - 4.9.1. Default Argument Values
 - 4.9.2. Keyword Arguments
 - 4.9.3. Special parameters
 - 4.9.3.1. Positional-or-Keyword Arguments
 - 4.9.3.2. Positional-Only Parameters
 - 4.9.3.3. Keyword-Only Arguments
 - 4.9.3.4. Function

4.1. **if** Statements

Perhaps the most well-known statement type is the **if** statement. For example:

```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...     x = 0
...     print('Negative changed to zero')
... elif x == 0:
...     print('Zero')
... elif x == 1:
...     print('Single')
... else:
...     print('More')
...
More
```

There can be zero or more **elif** parts, and the **else** part is optional. The keyword 'elif' is short for 'else if', and is useful to avoid excessive indentation. An **if ... elif ... elif ...** sequence is a substitute for the **switch** or **case** statements found in other languages.

If you're comparing the same value to several constants, or checking for specific types or attributes, you may also find the **match** statement useful. For more details see [match Statements](#).

4.2. **for** Statements

The **for** statement in Python differs a bit from what you may be used to in C or Pascal. Rather than always iter-

<https://docs.python.org/3/tutorial/>

- To learn about the way to define a function, read the sections 4.8 "Defining Functions" and 4.9 "More on Defining Functions".
 - `https://docs.python.org/3/tutorial/controlflow.html#defining-functions`
 - `https://docs.python.org/3/tutorial/controlflow.html#more-on-defining-functions`



"The Python Tutorial"

Table of Contents

- 4. More Control Flow Tools
 - 4.1. `if` Statements
 - 4.2. `for` Statements
 - 4.3. The `range()` Function
 - 4.4. `break` and `continue` Statements
 - 4.5. `else` Clauses on Loops
 - 4.6. `pass` Statements
 - 4.7. `match` Statements
 - 4.8. Defining Functions
 - 4.9. More on Defining Functions
 - 4.9.1. Default Argument Values
 - 4.9.2. Keyword Arguments
 - 4.9.3. Special parameters
 - 4.9.3.1. Positional-or-Keyword Arguments
 - 4.9.3.2. Positional-Only Parameters
 - 4.9.3.3. Keyword-Only Arguments
 - 4.9.3.4. Function Examples

4.8. Defining Functions

We can create a function that writes the Fibonacci series to an arbitrary boundary:

```
>>> def fib(n):    # write Fibonacci series less than n
...     """Print a Fibonacci series less than n."""
...     a, b = 0, 1
...     while a < n:
...         print(a, end=' ')
...         a, b = b, a+b
...     print()
...
... # Now call the function we just defined:
>>> fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 618 987 1597
```

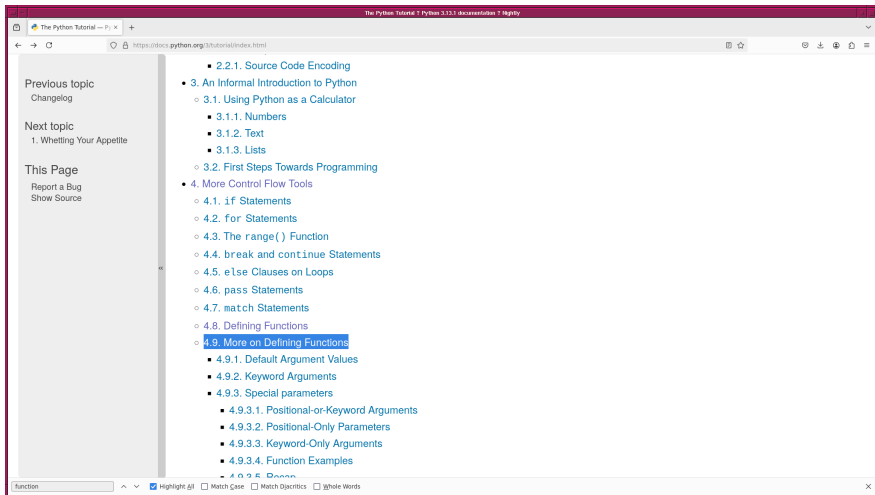
The keyword `def` introduces a function *definition*. It must be followed by the function name and the parenthesized list of formal parameters. The statements that form the body of the function start at the next line, and must be indented.

The first statement of the function body can optionally be a string literal; this string literal is the function's documentation string, or *docstring*. (More about docstrings can be found in the section [Documentation Strings](#).) There are tools which use docstrings to automatically produce online or printed documentation, or to let the user interactively browse through code; it's good practice to include docstrings in code that you write, so make a habit of it.

The *execution* of a function introduces a new symbol table used for the local variables of the function. More precisely, all variable assignments in a function store the value in the local symbol table; whereas variable references first look in the local symbol table, then in the local symbol tables of enclosing functions, then in the global symbol table, and finally in the table of built-in names. Thus, global variables and variables of enclosing functions

<https://docs.python.org/3/tutorial/>

"The Python Tutorial"



<https://docs.python.org/3/tutorial/>

"The Python Tutorial"

The screenshot shows a web browser displaying the Python documentation page for '4.9. More on Defining Functions'. The browser's address bar shows the URL <https://docs.python.org/3/tutorial/controlflow.html#more-on-defining-functions>. On the left, a 'Table of Contents' sidebar lists various topics, with '4.9. More on Defining Functions' and its sub-sections expanded. The main content area features the section title '4.9. More on Defining Functions', a paragraph explaining that functions can be defined with a variable number of arguments, and a sub-section '4.9.1. Default Argument Values'. This sub-section includes a paragraph about specifying default values and a code block for a function named `ask_ok`. Below the code block, it lists several ways to call the function: with only the mandatory argument, with one optional argument, or with all arguments.

4.9. More on Defining Functions

It is also possible to define functions with a variable number of arguments. There are three forms, which can be combined.

4.9.1. Default Argument Values

The most useful form is to specify a default value for one or more arguments. This creates a function that can be called with fewer arguments than it is defined to allow. For example:

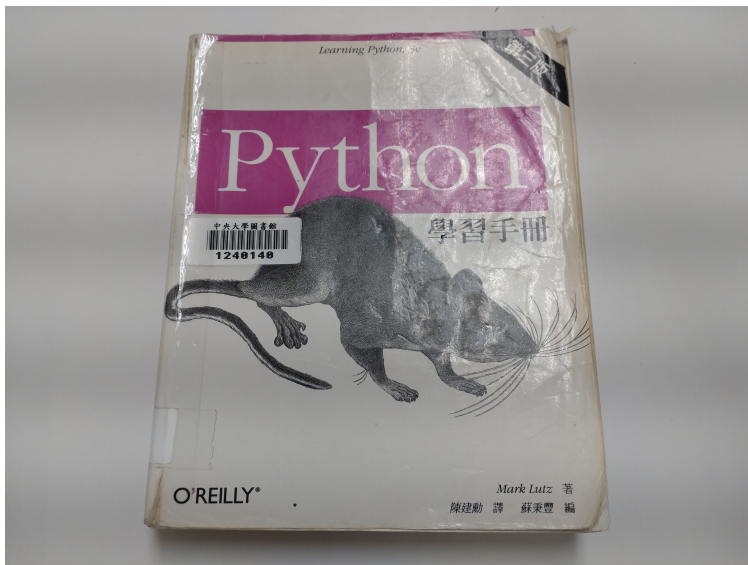
```
def ask_ok(prompt, retries=4, reminder='Please try again!'):
    while True:
        reply = input(prompt)
        if reply in {'y', 'ye', 'yes'}:
            return True
        if reply in {'n', 'no', 'nop', 'nope'}:
            return False
        retries = retries - 1
        if retries < 0:
            raise ValueError('invalid user response')
        print(reminder)
```

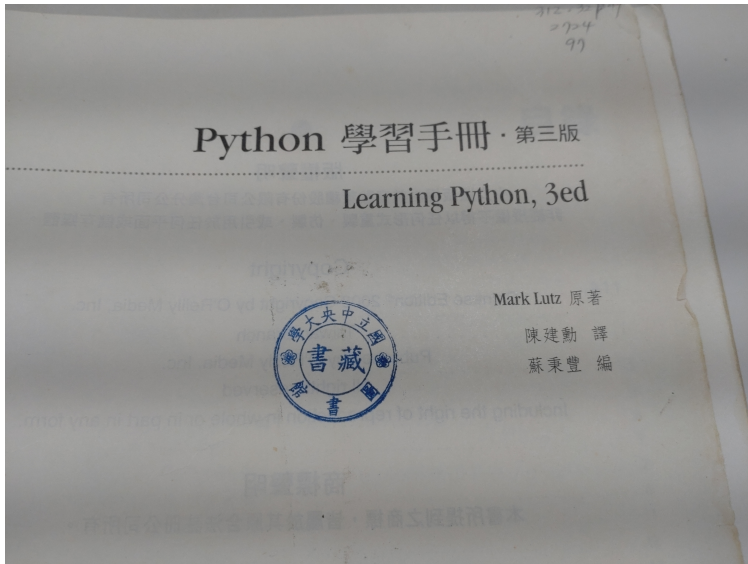
This function can be called in several ways:

- giving only the mandatory argument: `ask_ok('Do you really want to quit?')`
- giving one of the optional arguments: `ask_ok('OK to overwrite the file?', 2)`
- or even giving all arguments: `ask_ok('OK to overwrite the file?', 2, 'Come on, only yes or no!')`

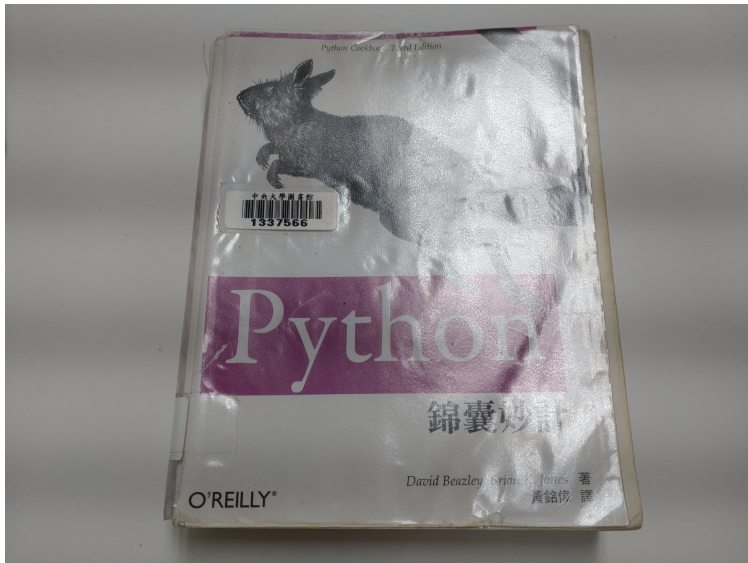
<https://docs.python.org/3/tutorial/>

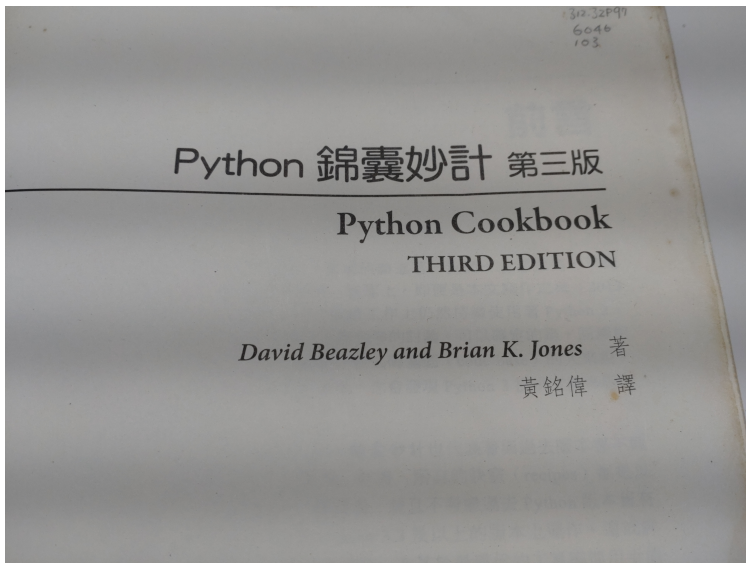
Recommended books



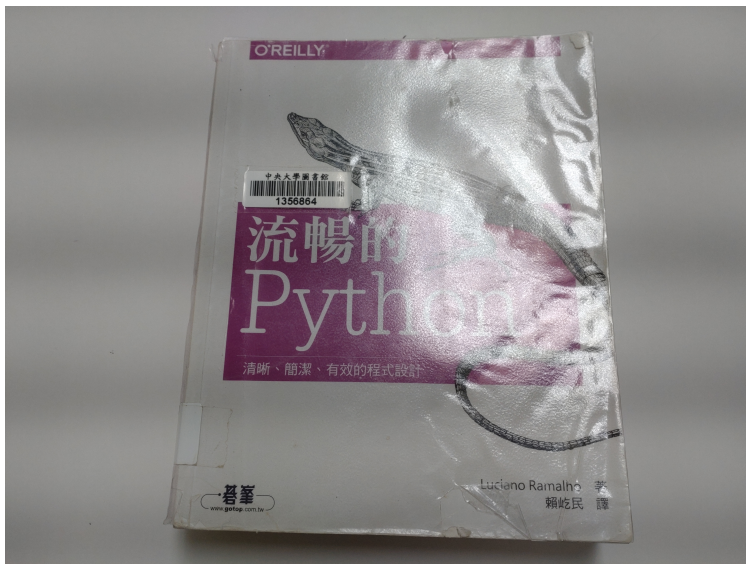


Recommended books





Recommended books



<https://www.instagram.com/daisuke23888/>

