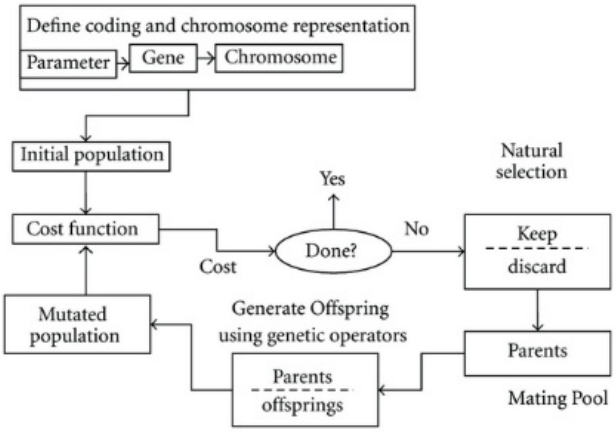
CHAPTER 3
ALGORITHMS AND PROGRAM DEVELOPMENT

Algorithm vs Program

- An ***algorithm*** **(演算法)** is a description of how to solve a problem
- A ***program*** **(程式)** is an implementation of an algorithm in a particular language to run on a computer (usually a particular kind of computer)
- Difference between ***what we want to do*** and ***what we actually did***

Algorithm vs Program

Algorithm: 通常用流程圖來表示演算法



Program: 將演算法轉換成程式，電腦才能執行

```
for v in activity_avi:
    avipath = vid_path.joinpath(v)
    if not vid_path.exists():
        print('no activity_avi:', str(avipath))
        continue

    vid_list = sorted(avipath.glob('*.avi'))

    dest_act_path = dest_path.joinpath(v)
    if not dest_act_path.exists():
        dest_act_path.mkdir()

    for f in vid_list:
        fname = f.name
        vid_title = fname.rsplit('.', 2)[0]
        vid_name = str(f)

        outpic_path = dest_act_path.joinpath(vid_title)
        if not outpic_path.exists():
            outpic_path.mkdir()
        else:
            shutil.rmtree(str(outpic_path))
            outpic_path.mkdir()

        cmd_str = 'ffmpeg  -i {} -r {} {}/{}.%4d.jpg  -loglev
```

Aspects of a Program: Readability

- Readability:程式可讀性要好, 讓人容易讀, 容易理解
- We will emphasize, over and over, that a program is an essay on problem solving intended to be read by other people, even if “other people” is you in the future!
- Write a program so that you can read it, because it is likely that sometime in the future ***you will*** have to read it!

Readability: Naming

- Naming (命名): variables, functions, modules, etc.
- The easiest thing to do that affects readability is good naming
  - use names for the items you create that reflect their purpose
  - to help keep straight the types used, include that as part of the name. Python does not care about the type stored, but you do!
  - remember "lower_with_under"

```
a = input("give a number: ")
b,c=1,0
while b<=a:
    c = c + b
    b = b + 1
print(a,b,c)
print("Result: ", c/b-1)
```

What are a, b, and c?

versus

```
# Calculate the average of a sum of consecutive integers in a given range.

limit_str=input("Range is from 1 to your input:")

limit_int = int(limit_str)
count_int = 1
sum_int = 0
while count_int <= limit_int:
        sum_int = sum_int + count_int
        count_int = count_int + 1
average_float = sum_int/(count_int - 1)
print("Average of sum of integers from 1 to",limit_int,"is", average_float)
```

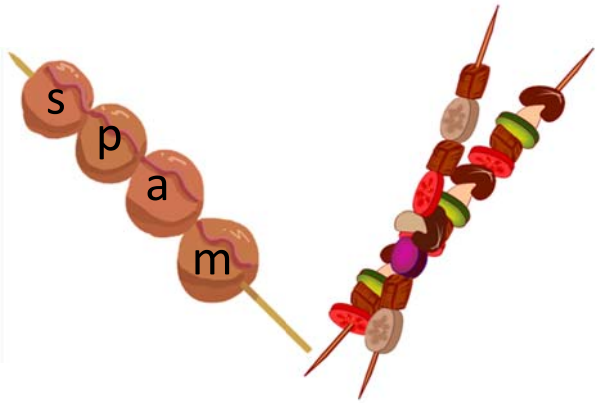
Readability: Comments

- info at the top, the goal of the code
- purpose of variables (if not obvious by the name)
- purpose of other functions being used
- anything ***tricky***. If it took you time to write, it probably is hard to read and needs a comment

CHAPTER 4
WORKING WITH STRINGS

Strings (字串)

- String is a sequence of characters.
- A string is indicated between Single Quotes ' ' or double quotes " "
  - `S = "spam"`
  - `s = 'spam'`
- Just don't mix them
  - `my_str = 'hi mom"` ⇒ ERROR



Strings

- Inserting an apostrophe ‘:
  - `A = "knight's"` # *mix up the quotes*
  - `B = 'knight\'s'` # *escape* single quote
- **Triple Quotes**
  - specify multi-line strings using triple quotes - (""" or ”’).
  - You can use single quotes and double quotes freely within the triple quotes.

```
'''This is a multi-line string. This is the first line.
This is the second line.
"What's your name?," I asked.
He said "Bond, James Bond."
'''
```

non-printing characters

- If inserted directly, are preceded by a backslash (the \ character)
  - new line `'\n'`
  - tab `'\t'`

String Representation

- Every character is "mapped" (associated) with an integer
- UTF-8 (ASCII), subset of Unicode, is such a mapping
- The function `ord()` takes a character and returns its UTF-8 integer value, `chr()` takes an integer and returns the UTF-8 character.

```
In [5]: ord('A')
Out[5]: 65

In [6]: chr(66)
Out[6]: 'B'
```

Char	Dec	Char	Dec	Char	Dec
SP	32	@	64	`	96
!	33	A	65	a	97
"	34	B	66	b	98
#	35	C	67	c	99
$	36	D	68	d	100
%	37	E	69	e	101

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

4.1.4 Strings as a sequence

- The Index
  - Because the elements of a string are a sequence, we can associate each element with an ***index***, a location in the sequence:
  - positive values count up from the left, beginning with index 0
  - negative values count down from the right, starting with -1

characters	H	e	l	l	o		W	o	r	l	d
index	0	1	2	3	4	5	6	7	8	9	10
									...	−2	−1

FIGURE 4.1 The index values for the string `'Hello World'`.

Accessing an element

- A particular element of the string is accessed by the index of the element surrounded by square brackets [ ]

```
hello_str = 'Hello World'
print(hello_str[1])  => prints e
print(hello_str[-1]) => prints d
print(hello_str[11]) => ERROR
```

characters	H	e	l	l	o		W	o	r	l	d
index	0	1	2	3	4	5	6	7	8	9	10
									...	−2	−1

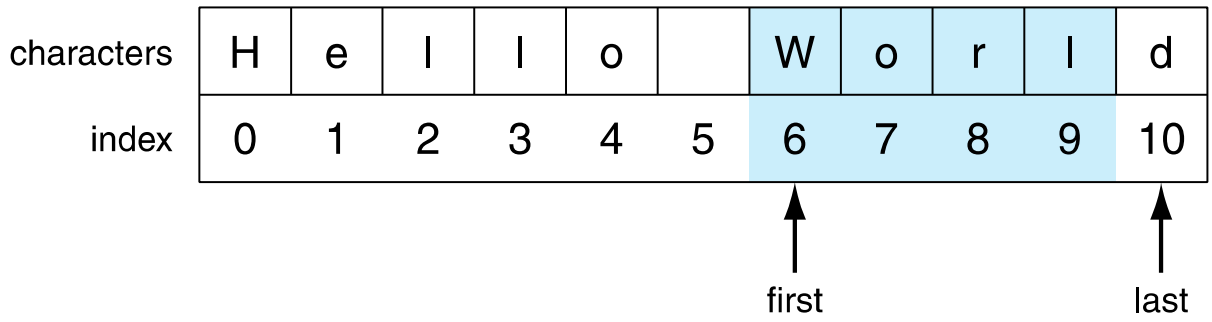
Slicing

- slicing is the ability to select a subsequence of the overall sequence
- uses the syntax `[start : finish]`, where:
  - `start` is the index of where we start the subsequence
  - `finish` is the index of **one after** where we end the subsequence
- if either `start` or `finish` are not provided, it defaults to the beginning of the sequence for `start` and the end of the sequence for `finish`

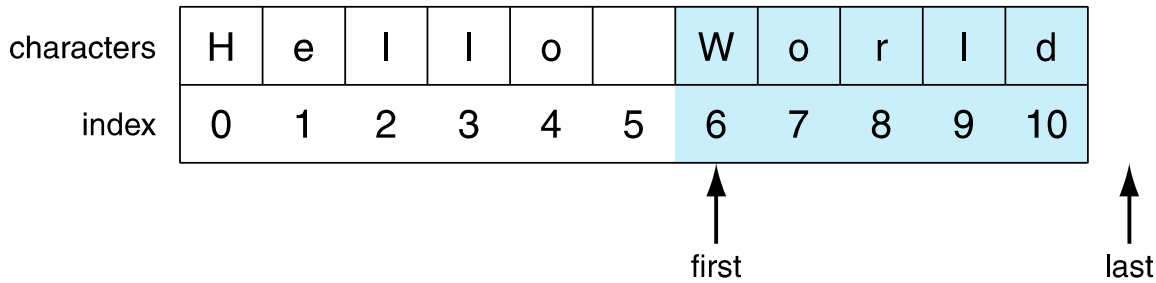
half open range for slices

- slicing uses what is called a half-open range
- the first index is included in the sequence
- the last index is one ***after*** what is included

```
helloString[6:10]
```




```
helloString[6:]
```




```
helloString[:5]
```

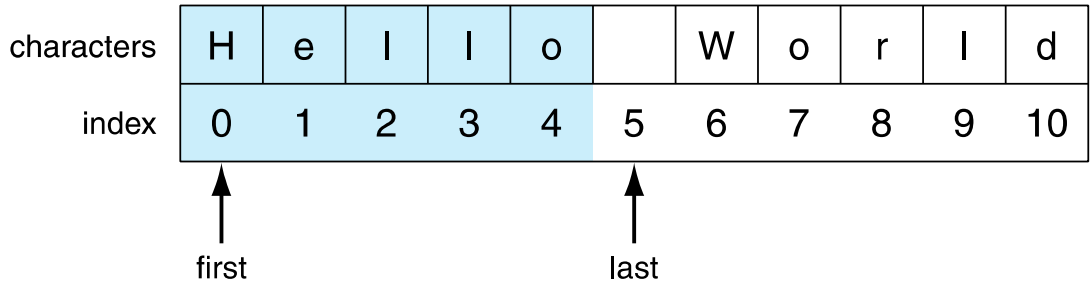


FIGURE 4.3 Two default slice examples.

`helloString[-1]`

Characters	H	e	l	l	o		W	o	r	l	d
Index	0	1	2	3	4	5	6	7	8	9	10
	−11	−10	−9	−8	−7	−6	−5	−4	−3	−2	−1

Last

`helloString[3:-2]`

Characters	H	e	l	l	o		W	o	r	l	d
Index	0	1	2	3	4	5	6	7	8	9	10

First Last

Exercise

- Assume the variable **`date`** has been set to a string value of the form *mm/dd/yyyy*, for example 09/08/2010. (Actual numbers would appear in the string.)
- Write a program to assign to a variable named **`dayStr`** the characters in date that contain the day. Then set a variable **`day`** to the integer value corresponding to the two digits in dayStr. Finally, output **`day`**.
  - That is, input “09/08/2010”, output 08

Exercise

- Assume that word is a variable of type String that has been assigned a value.
- Write a program whose value is a String consisting of the last three characters of the value of word.
- So if the value of word were **"biggest"** the output value would be **"est"**.

Extended Slicing

- also takes three arguments:
  - `[start:finish:step]`
- defaults are:
  - `start` is beginning, `finish` is end, `step` is 1

```
my_str = 'Hello world'
my_str[0:11:2] ⇒ 'HloWrd'
```

characters	H	e	l	l	o		W	o	r	l	d
index	0	1	2	3	4	5	6	7	8	9	10

Slicing with a step

- Every other letter

```
In [6]: hello_str = "Hello World"
In [7]: hello_str[::2] helloString[::2]
Out[7]: 'HloWrd'

In [8]: hello_str[::3]
Out[8]: 'HlWl'

In [9]: hello_str[::-1]
Out[9]: 'dlroW olleH'

In [10]: hello_str[::-2]
Out[10]: 'drWolH'
```

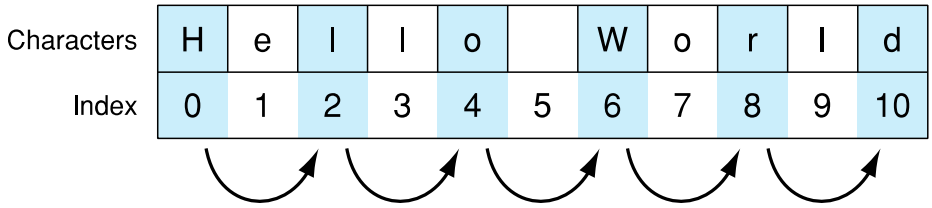


FIGURE 4.6 Slicing with a step.

Slicing with a step

characters	0	1	2	3	4	5	6	7	8	9
index	0	1	2	3	4	5	6	7	8	9

```
In [11]: digits = "0123456789"
In [12]: digits[::2]
Out[12]: '02468'

In [13]: digits[1::2]
Out[13]: '13579'

In [14]: digits[::-1]
Out[14]: '9876543210'

In [15]: digits[::-2]
Out[15]: '97531'

In [16]: digits[-2::-2]
Out[16]: '86420'
```

Copy and reverse a string

- Copy a string

```
In [17]: my_str = 'hi mom'
In [18]: new_str = my_str[:]
In [19]: new_str
Out[19]: 'hi mom'
```

```
In [9]: my_str = 'hi mom'
        new_str = my_str[:]
        if my_str is new_str:
            print('same object')
        else:
            print('different object')

same object
```

Sequences are iterable (可迭代)

- iterates (迭代)
  - to examine each individual element
  - iterable (可迭代): 可以一個一個檢查跑過
- The for loop iterates (迭代) through each element of a sequence in order. For a string, this means character by character.

```
>>> for char in 'Hi mom':
        print(char, type(char))

H <class 'str'>
i <class 'str'>
  <class 'str'>
m <class 'str'>
o <class 'str'>
m <class 'str'>
>>>
```

4.2 String Operations

- Basic String Operations
  - `s = 'spam'`
  - length operator len()
    - `len(s)` ⇒ 4
  - + is concatenate
    - `new_str = 'spam' + '-' + 'spam-'`
    - `print(new_str)` ⇒ `spam-spam-`
  - * is repeat, the number is how many times
    - `new_str * 3` ⇒
    - `'spam-spam-spam-spam-spam-spam-'`

Exercise

- Given three **string** variables that have been given values, **areaCode**, **exchange**, and **lastFour**, write a **string expression** whose value is the string equivalent of each these variables joined by a single hyphen (-)
- So if **areaCode**, **exchange**, and **lastFour**, had the values 800, 555, and 1212, the expression's value would be "800-555-1212".
- Alternatively, if **areaCode**, **exchange**, and **lastFour**, had the values 212, 867 and 5309 the expression's value would be "212-867-5309".

Exercise: alternating characters

- Given the strings s1 and s2, not necessarily of the same length, create a new string consisting of alternating characters of s1 and s2
  - that is, the first character of s1 followed by the first character of s2, followed by the second character of s1, followed by the second character of s2, and so on.
- Once the end of either string is reached, the remainder of the longer string is added to the end of the new string.
- For example, if s1 contained "abc" and s2 contained "uvwxyz", then the new string should contain "aubvcwxyz". Associate the new string with the variable s3.

Exercise: string permutation

- Input a string s, print all of the three-character permutations of s. Also print the total number of permutations.
- Note: assume the maximum length of the string s is 10

Example:
input a string:abcde
abc abd abe acb acd ace adb adc ade aeb aec aed bac bad bae bca bcd
bce bda bdc bde bea bec bed cab cad cae cba cbd cbe cda cdb cde cea
ceb ced dab dac dae dba dbc dbe dca dcb dce dea deb dec eab eac ead
eba ebc ebd eca ecb ecd eda edb edc
The total number of permutations is 60

The type function

- You can check the type of the value associated with a variable using type
  - `my_str = 'hello world'`
  - `type(my_str)` ⇒ `<type 'str'>`
  - `my_str = 245`
  - `type(my_str)` ⇒ `<type 'int'>`

String comparisons: single char

- Python 3 uses the Unicode mapping for characters.
  - Allows for representing non-English characters
  - UTF-8, subset of Unicode, takes the English letters, numbers and punctuation marks and maps them to an integer.
  - Single character comparisons are based on that number

comparisons within sequence

- It makes sense to compare within a sequence (lower case, upper case, digits).
  - `'a' < 'b'` → `True`
  - `'A' < 'B'` → `True`
  - `'1' < '9'` → `True`
- Can be weird outside of the sequence
  - `'a' < 'A'` → `False`
  - `'a' < '0'` → `False`

Comparisons with more than one char

- Compare the first element of each string
  - if they are equal, move on to the next character in each
  - if they are not equal, the relationship between those to characters are the relationship between the string
  - if one ends up being shorter (but equal), the shorter is smaller

Comparison examples

- `'a' < 'b'` ➔ `True`
- `'aaab' < 'aaac'`
  - first difference is at the last char. `'b'<'c'` so `'aaab'` is less than `'aaac'`. ➔ `True`
- `'aa' < 'aaz'`
  - The first string is the same but shorter. Thus it is smaller. ➔ `True`
- `'ba' < 'aaz'` ➔ `False`

Membership operations: in

- The **in** operator can check to see if a substring exists in the string. Returns True or False
  - `my_str = 'aabbccdd'`
  - `'a' in my_str` ➔ `True`
  - `'a' not in my_str` ➔ `False`
  - `'abb' in my_str` ➔ `True`
  - `'x' in my_str` ➔ `False`

Strings are immutable

- strings are immutable, that is you cannot change one once you make it:
  - `a_str = 'spam'`
  - `a_str[1] = 'l'` → ERROR
- However, you can use it to make another string (copy it, slice it, etc.)
  - `new_str = a_str[:1] + 'l' + a_str[2:]`
  - `a_str` → 'spam'
  - `new_str` →'slam'

Exercise

- Given a variable s that is associated with non-empty string, write some **statements** that use a **while or for loop** to associate a variable vowel_count with the number of lower-case vowels ("a","e","i","o","u") in the string.
  - 輸出字串裡面母音字母的個數

4.3 STRING METHODS AND FUNCTIONS

Functions

- A function is a program that performs some operation. Its details are hidden (encapsulated), only it's interface provided.
- A function takes some number of inputs (arguments) and returns a value based on the arguments and the function's operation.
- String function: **`len()`**
  - The **`len()`** function takes as an argument a string and returns an integer, the length of a string.

```
my_str = 'Hello World'
len(my_str) ➔ 11 # space counts!
```

String method

- A ***method*** is a variation on a function
  - like a function, it represents a program
  - like a function, it has input arguments and an output
- Unlike a function, it is applied in the context of a particular object.
- This is indicated by the *dot notation* invocation
- **`upper()`** is the name of a method.
  - It generates a new string that has all upper case characters of the string it was called with.

```
my_str = 'Python Rules!'
my_str.upper() ➔ 'PYTHON RULES!'
```

more dot notation

- Dot notation looks like: **`object.method(…)`**
- It means that the object in front of the dot is calling a method that is associated with that object's type.
- The method's that can be called are tied to the type of the object calling it. Each type has different methods.
- **`find(),index()`** method

```
In [1]: my_str = 'hello'
In [2]: my_str.index('e')
Out[2]: 1

In [3]: my_str.index('ll')
Out[3]: 2
In [4]: my_str.find('ll')
Out[4]: 2
```

- len(my_str): len() is a function, not a method()
- find(), index() are methods.

The thing(s) in parenthesis, i.e. the 'll' in this case, is called an **argument**.

Chaining methods

- Methods can be chained together.
  - Perform first operation, yielding an object
  - Use the yielded object for the next method

```
my_str = 'Python Rules!'
my_str.upper() ➔ 'PYTHON RULES!'
my_str.upper().find('O')
➔ 4
```

Optional Arguments

- Some methods have optional arugments:
  - if the user doesn't provide one of these, a default is assumed
  - find has a default second argument of 0, where the search begins

```
a_str = 'He had the bat'
a_str.find('t') ⇒ 7  # 1st 't',start at 0
a_str.find('t',8) ⇒ 13  # 2nd 't'
```

Nesting Methods

- You can “nest” methods, that is the result of one method as an argument to another
- remember that parenthetical expressions are did “inside out”: do the inner parenthetical expression first, then the next, using the result as an argument

```
a_str.find('t', a_str.find('t')+1)
```

- translation: find the second 't'.

How to know?

- Goto python official site to find available methods for any type.
  - https://www.python.org/doc/
  - https://docs.python.org/3/library/string.html

https://docs.python.org/3/library/stdtypes.html#string-methods

capitalize( )	lstrip([*chars*])
center(*width* [, *fillchar*])	partition(*sep*)
count(*sub* [, *start* [, *end*]])	replace(*old*, *new* [, *count*])
decode([*encoding* [, *errors*]])	rfind(*sub* [,*start* [,*end*]])
encode([*encoding* [,*errors*]])	rindex(*sub* [, *start* [, *end*]])
endswith(*suffix* [, *start* [, *end*]])	rjust(*width* [, *fillchar*])
expandtabs([*tabsize*])	rpartition(*sep*)
find(*sub* [, *start* [, *end*]])	rsplit([*sep* [,*maxsplit*]])
index(*sub* [, *start* [, *end*]])	rstrip([*chars*])
isalnum( )	split([*sep* [,*maxsplit*]])
isalpha( )	splitlines([*keepends*])
isdigit( )	startswith(*prefix* [, *start* [, *end*]])
islower( )	strip([*chars*])
isspace( )	swapcase( )
istitle( )	title( )
isupper( )	translate(*table* [, *deletechars*])
join(*seq*)	upper( )
lower( )	zfill(*width*)
ljust(width [, *fillchar*])	

TABLE 4.2 Python String Methods

method specification

- I highly recommend that you go to the python official website to learn about the method specification.
  - https://docs.python.org/3/library/stdtypes.html#string-methods
- For example, **`str.find(sub[, start[, end]])`**
  - Return the lowest index in the string where substring sub is found within the slice s[start:end].
  - Optional arguments start and end are interpreted as in slice notation.
  - Return -1 if sub is not found.

Exercise

- Given variables **first** and **last**, each of which is associated with a str, representing a first and a last name, respectively. Write an **expression** whose value is a str that is a full name of the form "Last,First".
- So, if **first** were associated with "alan" and **last** with "turing", then your expression would be "Turing,Alan".
  - (Note the capitalization! Note: no spaces!)
- Hint: capitalize()

Exercise

- Write an **expression** whose value is the same as the str associated with s but with **reversed case**. (大寫換小寫，小寫換大寫)
  - Thus, if the str associated with s is "McGraw", the value of the expression would be "mCgRAW".
  - Hint: swapcase()

Exercise

- Input a string and output the string type
  - if all characters in the string are alphabetic, output “alphabetic”
  - if all characters in the string are decimal characters, output “decimal”
  - if all characters in the string are hexadecimal (Base 16, 16進位) characters , output “hexadecimal ”
    - *hexadecimal* digit includes 0-9 plus A-F or a-f.

Exercise: Reverse Only Letters

- Write a function, given a string S, return the "reversed" string where all characters that are not a letter stay in the same place, and all letters reverse their positions.
- Note:
  - S.length <= 100
  - 33 <= S[i].ASCIIcode <= 122
  - S doesn't contain \ or “

```
Example 1:
    Input: "ab-cd"
    Output: "dc-ba"
Example 2:
    Input: "a-bC-dEf-ghIj"
    Output: "j-Ih-gfE-dCba"
Example 3:
    Input: "Test1ng-Leet=code-Q!"
    Output: "Qedo1ct-eeLg=ntse-T!"
```

strip method

- **`str.strip([chars]);`**
- **Parameters**
  - **chars** – The characters to be removed from beginning or end of the string.
  - The method **strip()** returns a copy of the string in which all chars have been stripped from the beginning and the end of the string (default whitespace characters).
- **Example**
  - str = "0000000this is string example....wow!!!0000000"
  - print(str.strip('0'))
  - **Output:** this is string example....wow!!!

Whitespace characters

- Whitespace characters include space(32), tab(9), linefeed(10), return(13), formfeed(12), and vertical tab(11).

```
import string
for c in string.whitespace:
    print(ord(c), end=' ')
```

```
32 9 10 13 11 12
```

ASCII 碼					ASCII 碼				
十進位	十六進位	字元	控制字元	意義	十進位	十六進位	字元	控制字元	意義
000	00		NULL	空字元	016	10	►	DLE	
001	01	☺	SOH		017	11	◄	DC1	
002	02	☻	STX		018	12	↕	DC2	
003	03	♥	ETX		019	13	‼	DC3	
004	04	♦	EOT		020	14	¶	DC4	
005	05	♣	ENQ		021	15	§	NAK	
006	06	♠	ACK		022	16	▬	SYN	
007	07	•	BELL	鈴聲	023	17	↨	ETB	
008	08	◘	BS	倒退鍵	024	18	↑	CAN	
009	09		HT	定位鍵	025	19	↓	EM	
010	0A		LF	line feed	026	1A	→	SUB	檔案結束
011	0B	♂	VT	home	027	1B	←	ESC	escape
012	0C	♀	FF	form feed	028	1C	∟	FS	向右游標
013	0D		CR	carriage return	029	1D	↔	GS	向左游標
014	0E	♫	SO		030	1E	▲	RS	向上游標
015	0F	☼	SI		031	1F	▼	US	向下游標

split method

- **`str.split(sep=None, maxsplit=-1)`**
  - Return a list of the words in the string, using sep as the delimiter string.
  - If maxsplit is given, at most maxsplit splits are done (thus, the list will have at most maxsplit+1 elements).
  - If maxsplit is not specified or -1, then there is no limit on the number of splits (all possible splits are made).
  - by default, if no argument is provided, split is on any whitespace character (tab, blank, etc.)

split method


```
>>> '1,2,3'.split(',')
['1', '2', '3']

>>> '1,2,3'.split(',', maxsplit=1)
['1', '2,3']

>>> '1,2,,3,'.split(',')
['1', '2', '', '3', '']
```

```
>>> '1 2 3'.split()
['1', '2', '3']

>>> '1 2 3'.split(maxsplit=1)
['1', '2 3']

>>> '   1   2   3   '.split()
['1', '2', '3']
```

The structure of a list


```
myList = [1, 'a', 3.14159, True]
```

myList

1	'a'	3.14159	True	
0	1	2	3	Index forward
−4	−3	−2	−1	Index backward

```
myList[1] → 'a'
myList[:3] → [1, 'a', 3.14159]
```

FIGURE 7.1 The structure of a list.

reorder a name


```
>>> name = 'John Marwood Cleese'
>>> first, middle, last = name.split()
>>> transformed = last + ', ' + first + ' ' + middle
>>> print(transformed)
Cleese, John Marwood
>>> print(name)
John Marwood Cleese
>>> print(first)
John
>>> print(middle)
Marwood
```

Exercise

- Write a program to input a sentence, and output the second word of the sentence.
  - For example: "This is a possible value of sentence."
  - The second word is "is" .

Exercise: Number of Segments in a String

- Count the number of segments in a string, where a segment is defined to be a contiguous sequence of non-space characters.
  - Please note that the string does not contain any **non-printable** characters.
- **Example 1**
  - Input: "Hello, my name is John"
  - Output: 5
- **Example 2**
  - Input: "Hello, fine"
  - Output: 2

Example 3

```
Input : "Hi, Hello, "
Output : 2
```

Example 4

```
Input : "I'm good.  "
Output : 2
```

Hint: using split()

Punctuation characters


```
import string
print(string.punctuation)
```

```
!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
```

- Exercise: removes punctuation from string
  - Input: hello, friend!... welcome.
  - Output: hello friend welcome

Exercise: Length of Last Word

- Given a string s consists of upper/lower-case alphabets and empty space characters ' ', return the length of last word in the string.
- If the last word does not exist, return 0.
- Note: A word is defined as a character sequence consists of non-space characters only, excluding punctuation characters.
- Example:
  - Input: “Hi World."
  - Output: 5

Exercise: Using split()

- Input an IP address string, determine whether it is a valid YZU IP.
  - A valid YZU IP address must be in the form of 140.138.xxx.xxx, where xxx is a number from 0-255. For example,
    - Invalid YZU IP: 140.138.21 192.111.54.251 140.138.100.300
    - Valid YZU IP: 140.138.41.110 140.138.100.175
  - You should check …
    - Does it contain four decimal numbers?
    - Is each number in the range 0-255?

Exercise: Using split()

- Input an email address string, determine whether it is a valid YZU email. A valid YZU email address must be in the form of username@xxx.yzu.edu.tw.
  - username and xxx are combinations of any character.
- Example:
  - Invalid YZU email address: s91110@yzu.edu.tw abc@yzu.edu.com
  - Valid YZU email address: s91110@mail.yzu.edu.tw catdog@iii.yzu.edu.tw

join method

- str.join(iterable)
  - Return a string which is the concatenation of the strings in iterable.
  - A TypeError will be raised if there are any non-string values in iterable, including bytes objects.
  - The separator between elements is the string providing this method.

```
str = "-"
seq = ("a", "b", "c")
print(str.join( seq ))

Output:
a-b-c
```

Exercise

- Write a program which accepts a sequence of comma separated 4 digit binary numbers as its input and then check whether they are divisible by 5 or not. The numbers that are divisible by 5 are to be printed in a comma separated sequence. Hint: int('0100',**2**) can convert binary string to int.
- Example:
  - Input : 0100,0011,1010,1001,1111
  - Then the output should be: 1010,1111

Exercise

- Displaying a Sentence with Its Words Reversed
  - Write a program that inputs a line of text, tokenizes the line and outputs the tokens in reverse order.
  - For example, input “today is hot”
  - Output “hot is today”

Exercise: Reverse Words in a String

- Given a string, you need to reverse the order of characters in each word within a sentence while still preserving whitespace and initial word order.
  - Ex. Today is hot ➔ yadoT si toh
- **Note:** In the string, each word is separated by single space and there will not be any extra space in the string.

4.4 Formatted output for strings

- String formatting, better printing
  - So far, we have just used the defaults of the print function
  - We can do many more complicated things to make that output "prettier" and more pleasing.
  - We will use it in our display function

Basic form

- To understand string formatting, it is probably better to start with an example.

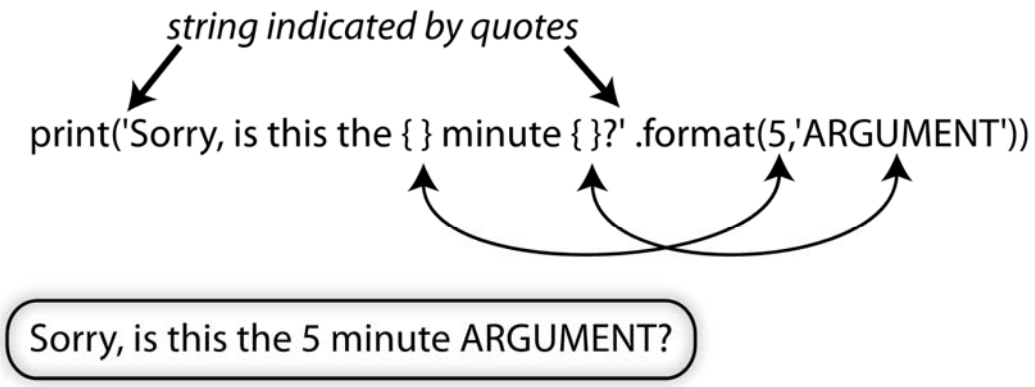


FIGURE 4.10 String formatting example.

format method

- **`format()`** is a method that creates a new string where certain elements of the string are re-organized
- map args to **`{}`**
  - The string is modified so that the **`{}`** elements in the string are replaced by the format method arguments
  - The replacement is in order: first **`{}`** is replaced by the first argument, second **`{}`** by the second argument and so forth.

Structure of the format command

- Each brace structure looks like

```
{:[align] [width] [.precision] [descriptor]}
```

  - Square brackets, [], indicate optional arguments.
  - **`align`** is optional (default left)
  - **`width`** is how many spaces (default just enough)
  - **`.precision`** is for floating point rounding (default no rounding)
  - **`descriptor code`** is the expected type (error if the arg is the wrong type)
  - All the optional information comes after a colon in the braces.

<	left
>	right
^	center

numbers indicate total spaces

print('{:>10s} is {:<10d} years old.' format('Bill', 25))

String 10 spaces wide including the object, right justified (>).

Decimal 10 spaces wide including the object, left justified (<).

OUTPUT:

Bill is 25 years old.

10 spaces 10 spaces

s	string
d	decimal integer
f	floating-point decimal
e	floating-point exponential
%	floating-point as percent

FIGURE 4.11 String formatting with width descriptors and alignment.

Nice table


```
>>> for i in range(5):
        print("{:10d} --> {:4d}".format(i,i**2))

         0 -->    0
         1 -->    1
         2 -->    4
         3 -->    9
         4 -->   16
```

Floating Point Precision

- Can round floating point to specific number of decimal places

```
>>> import math
>>> print(math.pi)                          # unformatted printing
3.141592653589793
>>> print("Pi is {:.4f}".format(math.pi)) # floating-point precision 4
Pi is 3.1416
>>> print("Pi is {:8.4f}".format(math.pi)) # specify both precision and width
Pi is   3.1416
>>> print("Pi is {:8.2f}".format(math.pi))
Pi is     3.14
```


4.5 CONTROL AND STRINGS

iteration through a sequence

- To date we have seen the while loop as a way to iterate over a suite (a group of python statements)
- We briefly touched on the for statement for iteration, such as the elements of a list or a string
- for statement
  - We use the for statement to process each element of a list, one element at a time

```
for item in sequence:
  block
```

What **for** means


```
my_str='abc'
for c in my_str:
    print(c)
```

- first time through, c = 'a' (my_str[0])
- second time through, c='b' (my_str[1])
- third time through, c='c' (my_str[2])
- no more sequence left, for ends

Code Listing 4.1 Find a letter


```
# Our implementation of the find function. Prints the index where
# the target is found; a failure message, if it isn't found.
# This version only searches for a single character.

river = 'Mississippi'
target = input('Input a character to find: ')
for index in range(len(river)):          # for each index
    if river[index] == target:           # check if the target is found
        print("Letter found at index: ", index)  # if so, print the index
        break                            # stop searching
else:
    print('Letter',target,'not found in',river)
```

Enumerate (列舉) function

- The enumerate function prints out two values: the index of an element and the element itself
- Can use it to iterate through both the index and element simultaneously, doing dual assignment

Code Listings 4.2 find with enumerate


```
# Our implementation of the find function. Prints the index where
# the target is found; a failure message, if it isn't found.
# This version only searches for a single character.

river = 'Mississippi'
target = input('Input a character to find: ')
for index,letter in enumerate(river):          # for each index
    if letter == target:                       # check if the target is found
        print("Letter found at index: ", index)  # if so, print the index
        break                                  # stop searching
else:
    print('Letter',target,'not found in',river)
```

Palindromes (迴文) and the rules

- A palindrome is a string that prints the same forward and backwards
  - 由前往後，由後往前，都一樣
- same implies that:
  - case does not matter
  - punctuation and whitespace characters are ignored
- "Madam I'm Adam" is thus a palindrome

lower case and punctuation

- Every letter is converted using the `lower` method
- `import string`, brings in a series of predefined sequences (`string.digits, string.punctuation, string.whitespace`)
- We remove all non-wanted characters with the **`replace`** method. First arg is what to replace, the second the replacement.

Code Listing 4.4 Palindromes


```
# Palindrome tester
import string

original_str = input('Input a string:')
modified_str = original_str.lower()

bad_chars = string.whitespace + string.punctuation

for char in modified_str:
    if char in bad_chars:  # remove bad characters
        modified_str = modified_str.replace(char,'')

if modified_str == modified_str[::-1]: # it is a palindrome
    print(\
'The original string is:  {}\n\
 the modified string is: {}\n\
 the reversal is:        {}\n\
 String is a palindrome'.format(original_str, modified_str, modified_str[::-1]))
else:
    print(\
'The original string is:  {}\n\
 the modified string is: {}\n\
 the reversal is:        {}\n\
 String is not a palindrome'.format(original_str,modified_str,modified_str[::-1]))
```

More String Formatting

- We said a format string was of the following form:

  {:[align] [width] [.precision] [descriptor]}

- Well, it can be more complicated than that

  {arg : [[fill] align] [sign] [#][0][width][,] [.precision] [descriptor]}

- That's a lot, so let's look at the details

arg

- To over-ride the {}-to-argument matching we have seen, you can indicate the argument you want in the braces
  - if other descriptor stuff is needed, it goes behind the arg, separated by a :

```
>>> print('{0} is {2} and {0} is also {1}'.format('Bill',25,'tall'))
Bill is tall and Bill is also 25
```

<	left
>	right
^	center

TABLE 4.4 Width alignments.

fill, =

- Besides alignment, you can fill empty spaces with a fill character:
  - 0= fill with 0's
  - += fill with +

<	left
>	right
^	center

TABLE 4.4 Width alignments.

```
>>> print('{0:.>12s} | {1:0=+10d} | {2:->5d}'.format('abc',35,22))
.........abc | +000000035 | ---22
```

'=': Forces the padding to be placed after the sign (if any) but before the digits. This is used for printing fields in the form '+000000120'. This alignment option is only valid for numeric types.

sign

- + means a sign for positive and negative numbers
- - means a sign for negative only (default)
- space means space for positive, minus for negative

```
>>> print('{0:.>12s} | {1:0=+10d} | {2:->5d}'.format('abc',35,22))
.........abc | +000000035 | ---22
```

Example

- args are before the :, format after

```
>>> print('{0:.>12s} | {1:0=+10d} | {2:->5d}'.format('abc',35,22))
.........abc | +000000035 | ---22
```

- for example {1:0=+10d} means:
  - 1→ second (count from 0) arg of format, 35
  - :→ separator
  - 0= →fill with 0's
  - +→ plus or minus sign
  - 10d→ occupy 10 spaces decimal

, and 0

- # is complicated, but the simple version is that it forces a decimal point 0 forces fill of zero's (equivalent to 0=)
- , put commas every three digits

```
>>> print('{:#6.0f}'.format(3)) # decimal point forced
    3.
>>> print('{:04d}'.format(4)) # zero preceeds width
0004
>>> print('{:,d}'.format(1234567890))
1,234,567,890
```

nice for tables


```
>>> for n in range(3,11):
    print('{:4}-sides:{:6}{:10.2f}{:10.2f}'.format(n,180*(n-2),180*(n-2)/n,360/n)


   3-sides:   180     60.00    120.00
   4-sides:   360     90.00     90.00
   5-sides:   540    108.00     72.00
   6-sides:   720    120.00     60.00
   7-sides:   900    128.57     51.43
   8-sides:  1080    135.00     45.00
   9-sides:  1260    140.00     40.00
  10-sides:  1440    144.00     36.00
```