**فصل 11**

**دیکشنری ها**

این فصل یک نوع داخلی[[1]](#footnote-1) دیگر به نام دیکشنری را ارائه می کند. دیکشنری ها یکی از بهترین ویژگی های پایتون هستند؛ آنها زیر بنای خیلی از الگوریتم های کارآمد[[2]](#footnote-2) و ظریف[[3]](#footnote-3) هستند.

11. دیکشنری یک نگاشت[[4]](#footnote-4) است

دیکشنری مثل یک لیست هست اما عمومی تر. در لیست ها، اندیس ها باید عدد صحیح[[5]](#footnote-5) باشند؛ در یک دیکشنری آنها می توانند (تقریباً) هر نوعی باشند.

یک دیکشنری شامل مجموعه ای از اندیس هاست که به آنها **کلید[[6]](#footnote-6)** می گوییم و مجموعه ای از مقادیر[[7]](#footnote-7). هر کلید فقط به یک مقدار نسبت داده می شود. به یک کلید و یک مقدار، **زوج کلید-مقدار**[[8]](#footnote-8) یا گاهی به آنها **آیتم**[[9]](#footnote-9) گفته می شود.

به زبان ریاضی، یک دیکشنری نشان دهنده یک نگاشت از کلید ها به مقدار هاست، بنابراین می توان گفت هر کلید به یک مقدار «نگاشته می شود»[[10]](#footnote-10). به عنوان مثال، ما یک دیکشنری خواهیم ساخت که کلمات انگلیسی را به اسپانیایی می نگارد، بنابراین کلید ها و مقدار ها همه رشته[[11]](#footnote-11) هستند.

تابع dict یک دیکشنری جدید بدون هیچ آیتی می سازد. به این دلیل که dict نام یک تابع داخلی است، نباید از آن به عنوان اسم یک متغیر استفاده کنید.

>>> eng2sp = dict()

>>> eng2sp

{}

کروشه ها، {}، یک دیکشنری خالی رو نشان می دهند. برای اضافه کردن آیتم ها به دیکشنری، می توانید از براکت استفاده کنید:

>>> eng2sp['one'] = 'uno'

این خط یک آیتم می سازد که از کلید ‘one’ به مقدار ‘uno’ می نگارد. اگر ما دیکشنری رو دوباره چاپ کنیم، یک زوج کلید-مقدار با یک دونقطه بین کلید و مقدار می بینیم:

>>> eng2sp

{'one': 'uno'}

این فرمت خروجی، یک فرمت ورودی نیز هست. برای مثال، می توانید یک دیکشنری جدید با سه آیتم بسازید:

>>> eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}

اما اگر eng2sp را چاپ کنید، ممکن است قافل گیر شوید:

>>> eng2sp

{'one': 'uno', 'three': 'tres', 'two': 'dos'}

ترتیب زوج های کلید مقدار ممکن است یکسان نباشد. اگر همین مثال را در کامپیوترتان تایپ کنید، ممکن است جواب متفاوتی بگیرید. عموماً، ترتیب آیتم ها در یک دیکشنری غیر قابل پیش بینی است.

اما این مشکلی نیست، چون المان های یک دیکشنری هیچ گاه با اندیس های عدد صحیح جستجو نمی شوند. بلکه به جای آن، از کلید ها برای جستجو مقادیر متناظر استفاده می شود.

>>> eng2sp['two']

'dos'

کلید ‘two’ همیشه به مقدار ‘dos’ نگاشته می شود بنابراین ترتیب آیتم ها مهم نیست.

اگر آن کلید در دیکشنری نباشد، یک استثنا[[12]](#footnote-12) می گیرید:

>>> eng2sp['four']

KeyError: 'four'

تابع len بر روی دیکشنری ها کار می کند؛ تعداد زوج های کلید مقدار را بر می گرداند.

>>> len(eng2sp)

3

عملگرin نیز بر روی دیکشنری ها کار می کند. این عملگر به شما می گوید که آیا چیزی به عنوان یک کلید در دیکشنری ظاهر شده است. (به عنوان مقدار ظاهر شدن کافی نیست).

>>> 'one' in eng2sp

True

>>> 'uno' in eng2sp

False

برای اینکه ببینید چیزی به عنوان یک مقدار در دیکشنری ظاهر شده است، می توانید از متد values استفاده کنید، که مجموعه ای از مقادیر را بر می گرداند، و سپس از عملگر in استفاده کنید.

>>> vals = eng2sp.values()

>>> 'uno' in vals

True

عملگر in از الگوریتم های متفاوتی برای لیست ها و دیکشنری ها استفاده می کند. برای لیست ها، المان های لیست را به ترتیب جستجوی می کند، مانند بخش 8.6. هر چه لیست طولانی تر باشد، زمان جستجو با نسبت مستقیم طولانی تر می شود.

برای دیکشنری ها، پایتون از الگوریتمی به نام درهم ساز[[13]](#footnote-13) که خصوصیت قابل توجهی دارد: عمگر in بدون توجه به اینکه چند آیتم در دیکشنری است زمان یکسانی طول می کشد. در بخش B.4 توضیح می دهم که چگونه چنین چیزی امکان پذیر است، اما ممکن است توضیحات تا زمانی که چند فصل دیگر را نخوانید، قابل فهم نباشد.

**11.2 دیکشنری به عنوان مجموعه ای از شمارنده ها**

فرض کنید رشته ای به شما داده شده است و می خواهید بشمرید چند بار هر حرف تکرار شده است. راه های مختلفی برای انجام آن است:

1. می توانید 26 متغیر بسازید، یکی برای هر حرف الفبا. بعد می توانید رشته را پیشمایش کنید، برای هر کاراکتر، شمارنده متناظر با آن را افزایش دهید، احتمالاً از شرط های پشت سر هم استفاده کنید.
2. می توانید یک لیست با 26 المان درست کنید. بعد می توانید هر کاراکتر را به یک عدد تبدیل کنید(با استفاده از تابع داخلی ord)، از عدد به عنوان یک اندیس برای لیست استفاده کنید، و شمارنده مناسب را افزایش دهید.
3. می توانید یک دیکشنری با کاراکتر ها به عنوان کلید ها و شمارنده ها به عنوان مقدار های متناظراستفاده کنید. اولین باری که یک کاراکتر را می بینید، یک آیتم به دیکشنری اضافه می کنید. بعد از آن مقدار آیتمی که وجود دارد را اضافه می کنید.

هر کدام از این گزینه ها محاسبات یکسانی را انجام می دهند، اما هر کدام این محاسبات را به روش متفاوتی پیاده سازی می کنند:

یک **پیاده سازی[[14]](#footnote-14)** راهی برای انجام یک محاسبه[[15]](#footnote-15) است؛ بعضی از پیاده سازی ها بهتر از پیاده سازی های دیگر هستند. به عنوان مثال، یک مزیت پیاده سازی دیکشنری این است که نیازی نیست که از قبل بدانیم کدام حرف ها در رشته ظاهر شده اند و فقط نیاز است که برای حرف هایی که ظاهر می شوند فضا اختصاص بدهیم.

کد مورد نظر به شکل زیر خواهد بود:

def histogram(s):

d = dict()

for c in s:

if c not in d:

d[c] = 1

else:

d[c] += 1

return d

اسم تابع histogram است، که یک اصطلاح آماری برای یک مجموعه شمارنده(یا فراوانی) است.

اولین خط تابع یک دیکشنری خالی می سازد. حلقه for رشته را می پیماید. هر بار در حلقه، اگر کاراکتر c در دیکشنری نباشد، یک آیتم جدید با کلید c و مقدار اولیه 1 می سازیم. اگر c پیش از این در دیکشنری است d[c] را افزایش می دهیم.

به اینصورت کار می کند:

>>> h = histogram('brontosaurus')

>>> h

{'a': 1, 'b': 1, 'o': 2, 'n': 1, 's': 2, 'r': 2, 'u': 2, 't': 1}

هیستوگرام نشان می دهد که حرف های ‘a’ و ‘b’ یکبار تکرار شده اند. ‘o’ دوبار ظاهر شده است و....

دیکشنری ها متدی به نام get دارند که یک کلید و یک مقدار پیش فرض می گیرد. اگر کلید در دیکشنری باشد، get مقدار متناظر را برمی گرداند. در غیر اینصورت مقدار پیش فرض را بر می گرداند. برای مثال:

>>> h = histogram('a')

>>> h

{'a': 1}

>>> h.get('a', 0)

1

>>> h.get('b', 0)

0

به عنوان تمرین، از get برای نوشتن histogram به صورت خلاصه تر استفاده کنید. باید بتوانید که دستور if را حذف کنید.

11.3 حلقه ها و دیکشنری ها

اگر از یک دیکشنری در دستور for استفاده کنید، کلید های دیکشنری را پیمایش می کند. برای مثال، print\_hist هر کلید و مقدار متناظر را چاپ می کند:

def print\_hist(h):

for c in h:

print(c, h[c])

خروجی به صورت زیر خواهد بود:

>>> h = histogram('parrot')

>>> print\_hist(h)

a 1

p 1

r 2

t 1

o 1

دوباره، کلید ها هیچ ترتیب خاصی ندارند. دیکشنری ها متدی به نام keys دارند که کلید های دیکشنری را بدون هیچ ترتیب خاصی، به صورت یک لیست برمی گرداند. به عنوان تمرین، print\_hist را تغییر دهید تا کلید ها و مقدارهای آنها را به ترتیب حروف الفبا چاپ کند.

**11.4 جستجوی معکوس**

با یک دیکشنری d و یک کلید k، پیدا کردن مقدار متناظر d[k] = v آسان است. این عمل یک جستجو[[16]](#footnote-16) نامیده می شود.

اما اگر v را داشته باشید و بخواهید k را پیدا کنید چه می کنید؟ دو مشکل دارید: اولاً، ممکن است بیشتر از یک کلید باشد که به مقدار v نگاشته شود. بر حسب کاربرد، ممکن است که بتوانید یکی را انتخاب کنید، یا ممکن است نیاز باشد که لیستی بسازید که شامل همه آنها باشد. دوماً، هیچ ساختار ساده ای برای انجام یک **جستجو مکعوس[[17]](#footnote-17)** وجود ندارد. باید جستجو کنید.

تابع زیر یک مقدار می گیرد و اولین کلیدی که به آن مقدار نگاشته می شود بر می گرداند:

def reverse\_lookup(d, v):

for k in d:

if d[k] == v:

return k

raise LookupError()

این تابع مثال دیگری از الگوی جستجو است، اما از ویژگی استفاده می کند که ما قبلاً ندیده ایم، raise. دستور raise یک استثنا ایجاد می کند؛ در این مورد موجب یک LookupError می شود، که یک استثنا داخلی است که برای نشان دادن شکست یک عمل جستجو، استفاده می شود.

اگر به آخر حلقه برسیم، به این معنی است که v در داخل دیکشنری به عنوان مقدار وجود ندارد، بنابراین یک استثنا بر می انگیزیم[[18]](#footnote-18).

این یک مثال از یک جستجو معکوس موفق است:

>>> h = histogram('parrot')

>>> k = reverse\_lookup(h, 2)

>>> k

'r'

و یک جستجوی غیر موفق:

>>> k = reverse\_lookup(h, 3)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

File "<stdin>", line 5, in reverse\_lookup

ValueError

تاثیر برانگیختن استثنا به همانند زمانی است که پایتون بر می انگیزد: یک ردیابی[[19]](#footnote-19) و یک پیام خطا چاپ می کند.

دستور raise می تواند یک پیام خطا دقیق به عنوان آرگومان اختیاری بگیرد. برای مثال:

>>> raise ValueError('value does not appear in the dictionary')

Traceback (most recent call last):

File "<stdin>", line 1, in ?

ValueError: value does not appear in the dictionary

یک جستجو معکوس خیلی کند تر از یک جستجو (مستقیم[[20]](#footnote-20)) است؛ اگر باید اغلب آن را انجام دهید، یا اگر دیکشنری بزرگ شود، کارایی برنامتان کم خواهد شد.

11.5 **دیکشنری و لیست ها**

لیست ها می توانند به عنوان مقدار ها در دیکشنری ها ظاهر شوند. برای مثال، اگر یک دیکشنری به شما داده شود که حرف ها را به تعداد می نگارد، شاید شما بخواهید که بر عکسش کنید؛ یعنی، بخواهید یک دیکشنری بسازید که تعداد تکرار را به حرف ها می نگارد. به این دلیل که ممکن است حرف های زیادی با تعداد تکرار یکسان باشند، هر مقدار در دیکشنری معکوس باید لیستی از حرف ها باشد.

تابع زیر یک دیکشنری را معکوس می کند:

def invert\_dict(d):

inverse = dict()

for key in d:

val = d[key]

if val not in inverse:

inverse[val] = [key]

else:

inverse[val].append(key)

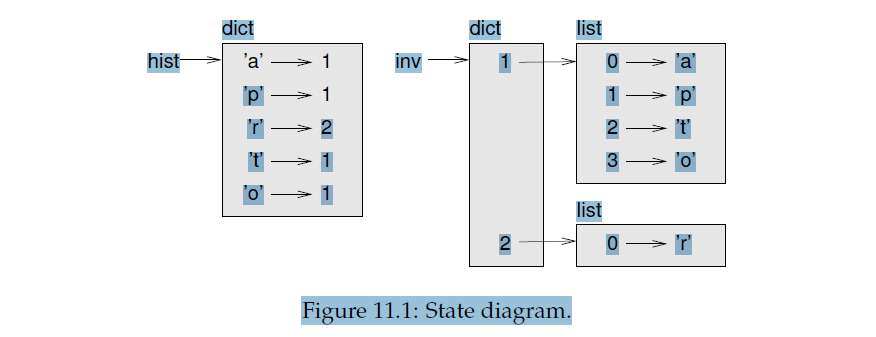
return inverse

هر بار در حلقه، key یک کلید از d و val مقدار متناظر آن را می گیرد. اگر val در inverse نباشد، به این معنی است که ما قبلاً آن را ندیده ایم، بنابراین، یک آیتم جدید می سازیم و با یک یگانه[[21]](#footnote-21)(لیستی با یک المان) مقدار دهی اولیه می کنیم. در غیر اینصورت ما این مقدار را قبلاً دیده ایم، بنابراین کلید متناظر با آن را به لسیت اضافه می کنیم.

به عنوان مثال:

>>> hist = histogram('parrot')

>>> hist



شکل 11.1 نمودار حالت

{'a': 1, 'p': 1, 'r': 2, 't': 1, 'o': 1}

>>> inverse = invert\_dict(hist)

>>> inverse

{1: ['a', 'p', 't', 'o'], 2: ['r']}

شکل 11.1 یک نمودار حالت است که hist و inverse را نمایش می دهد. یک دیکشنری به عنوان یک جعبه با نوع type بالای آن و زوج های کلید مقدار درون آن نمایش داده شده است. اگر مقدار ها عدد صحیح هستند، اعشاری یا رشته، آنها را درون جعبه می کشم، اما معمولاً لیست ها را بیرون جعبه می کشم تا نمودار ساده باشد.

لیست ها می توانند مقدار های درون یک دیکشنری باشند، همانطور که این مثال نشان می دهد، اما نمی توانند کلید باشند. اگر امتحان کنید این اتفاق می افتد:

>>> t = [1, 2, 3]

>>> d = dict()

>>> d[t] = 'oops'

Traceback (most recent call last):

File "<stdin>", line 1, in ?

TypeError: list objects are unhashable

من قبلاً گفته ام که یک دیکشنری با استفاده از یک درهم ساز پیاده سازی می شود و آن به این معنی است که کلید ها باید **قابل درهم سازی**[[22]](#footnote-22) باشد.

یک **درهم ساز**[[23]](#footnote-23) تابعی است که یک مقدار(از هر نوعی) می گیرد و یک عدد صحیح برمی گرداند. دیکشنری ها از این عدد صحیح ها که به آنها مقدار درهم ساز گفته می شود برای ذخیره سازی و جستجو زوج کلید مقدار ها استفاده می کنند.

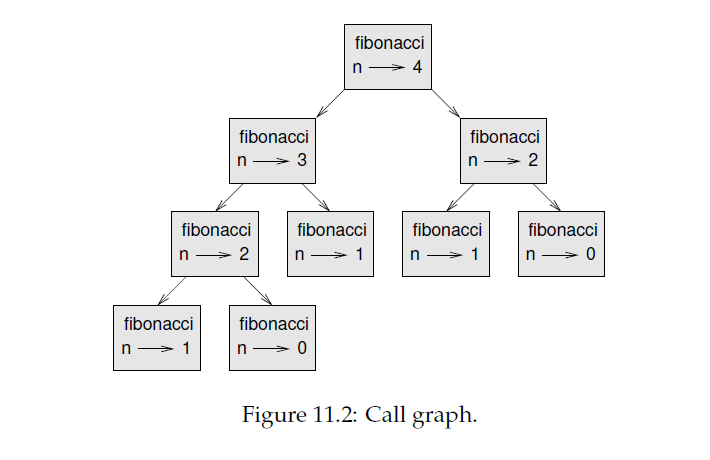
این سیستم اگر کلید ها غیر قابل تغییر[[24]](#footnote-24) باشند به خوبی کار می کند. اما اگر کلید ها قابل تغییر[[25]](#footnote-25) باشند، مانند لیست ها، اتفاق های بدی می افتد. برای مثال، وقتی یک زوج کلید مقدار می سازید، پایتون کلید را درهم سازی می کند و در جایگاه متناظر ذخیره می کند. اگر کلید را تغییر دهید و دوباره آن را درهم سازی کنید، به جای متفاوتی خواهد رفت. در اینصورت ممکن است دو ورودی برای کلید یکسان داشته باشید، یا ممکن است نتوانید کلید را پیدا کنید. در هر صورت، دیکشنری درست کار نخواهد کرد.

به همین دلیل است که کلید ها باید قابل درهم سازی باشند، و به همین دلیل است که نوع های قابل تغییر مانند لیست قابل درهم سازی نیستند. ساده ترین راه برای رفع این محدودیت استفاده از چندتایی[[26]](#footnote-26) ها است، که در فصل بعد می بینید.

به این دلیل که دیکشنری ها قابل تغییر هستند، به عنوان کلید نمی توان از آنها استفاده کرد، اما از آنها می توان به عنوان مقدار استفاده کرد.

11.6 Memos

اگر با تابع fibonacci از بخش 6.7 کارکرده اید، ممکن است متوجه شده باشید که هر چه آرگومان بزرگتری ارائه کنید، تابع زمان طولانی تری برای اجرا خواهد گرفت. علاوه بر این، زمان اجرا به سرعت زیاد می شود.



شکل 11.2: گراف فراخوانی

برای اینکه بفهمیم چرا، شکل 11.2 را در نظر بگیرید که گراف فراخوانی[[27]](#footnote-27) را برای fibonacci با n=4 نمایش می دهد:

یک گراف فراخوانی مجموعه ای از فریم های تابعی را نمایش می دهد.

1. built-in [↑](#footnote-ref-1)
2. efficient [↑](#footnote-ref-2)
3. elegant [↑](#footnote-ref-3)
4. mapping [↑](#footnote-ref-4)
5. integers [↑](#footnote-ref-5)
6. keys [↑](#footnote-ref-6)
7. values [↑](#footnote-ref-7)
8. key-value pair [↑](#footnote-ref-8)
9. item [↑](#footnote-ref-9)
10. maps [↑](#footnote-ref-10)
11. strings [↑](#footnote-ref-11)
12. exception [↑](#footnote-ref-12)
13. hashtable [↑](#footnote-ref-13)
14. implementation [↑](#footnote-ref-14)
15. computation [↑](#footnote-ref-15)
16. lookup [↑](#footnote-ref-16)
17. reverse lookup [↑](#footnote-ref-17)
18. raise an exception [↑](#footnote-ref-18)
19. traceback [↑](#footnote-ref-19)
20. forward-lookup [↑](#footnote-ref-20)
21. singleton [↑](#footnote-ref-21)
22. hashable [↑](#footnote-ref-22)
23. hash [↑](#footnote-ref-23)
24. immutable [↑](#footnote-ref-24)
25. mutable [↑](#footnote-ref-25)
26. tuple [↑](#footnote-ref-26)
27. call graph [↑](#footnote-ref-27)