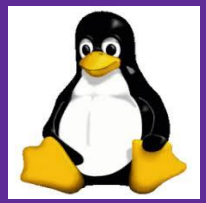
Introduction to Computation

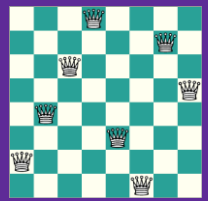
Autumn, 2023

Prof. Fan Cheng

Shanghai Jiao Tong University

chengfan85@gmail.com
https://github.com/ichengfan/itc





13

Outline

- Iterable
- Generator

Iterable

Iterable (可迭代)

Technically, in Python, an iterator is an object which implements the iterator protocol, which consist of the methods __iter__() and __next__()

for x in collection_a:

- In python, for could be used to loop over a collection_a, whose type may be

  list, tuple, str, dict, set, range, etc..

- An iterator is an object that can be iterated upon, meaning that you can traverse through all the values
- a virtual index for x: loop over them from left to right
  - Lists, tuples, dictionaries, and sets are all iterable objects
  - They are iterable containers which you can get an iterator from
  - All these objects have an iter() method to get an iterator
- Iter() – __iter__(), 初始化iterator; next() – __next__(), 获得迭代器的下一个对象

- 一个类，如果实现了__iter__(), __next()__函数，则可以通过iter()来获得它的迭代器，从而配合next()函数遍历它，实现连续访问
- 和这个类对应的iterable (iterator): 迭代器、广义下标. 可以用for来遍历
- 回顾：for x in collection_a: 给定后，修改x不改变collection_a和迭代顺序
- iter()+next() 获取collection_a的迭代顺序

iter(), next()

```python
ilst = iter([-1, 1, -2, 3, 4])
print(next(ilst))
print(next(ilst))
print(next(ilst))
print(next(ilst))
print(next(ilst))

itp = iter(tuple(range(4)))
print(next(itp))
print(next(itp))
print(next(itp))
print(next(itp))

str1 = "SJTU"
istr = iter("SJTU")
print(next(istr))
print(next(istr))
print(next(istr))
print(next(istr))

idt = iter({123: 123, -1: -2, -3: -5, 4.1: 3.14})
print(next(idt))
print(next(idt))
print(next(idt))
print(next(idt))
```

```
-1
1
-2
3
4
0
1
2
3
S
J
T
U
123
-1
-3
4.1
```

```python
ist = iter(set((-1, -2, -3, -4)))
print(next(ist))
print(next(ist))
print(next(ist))
print(next(ist))

ir = iter(range(-4, 5, 2))
print(next(ir))
print(next(ir))
print(next(ir))
print(next(ir))
print(next(ir))

M = [
    [1, 2, 3],  # A 3 × 3 matrix, as nested lists
    [4, 5, 6],  # Code can span lines if bracketed
    [7, 8, 9],
]

g = iter([sum(row) for row in M])
print(next(g))
print(next(g))
print(next(g))
```

```
-4
-3
-1
-2
-4
-2
0
2
4
6
15
24
```

iterable

- In a nutshell, an object is iterable if it is either a physically stored sequence in memory, or an object that generates one item at a time in the context of an iteration operation—a sort of "virtual" sequence
- More formally, both types of objects are considered iterable because they support the iteration protocol—they respond to the iter call with an object that advances in response to next calls and raises an exception when finished producing values
  - The generator comprehension expression we saw earlier is such an object: its values aren't stored in memory all at once, but are produced as requested, usually by iteration tools
  - Python file objects similarly iterate line by line when used by an iteration tool: file content isn't in a list, it's fetched on demand. Both are iterable objects in Python—a category that expands in 3.X to include core tools like range and map
- I'll have more to say about the iteration protocol later in this book. For now, keep in mind that every Python tool that scans an object from left to right uses the iteration protocol
  - This is why the sorted call used in the prior section works on the dictionary directly—we don't have to call the keys method to get a sequence because dictionaries are iterable objects, with a next that returns successive keys

--Learning Python, p.120

iterator VS. iterable

The iterator of an iterator is itself

```
1  r = range(-4, 5, 2)
2  ir = iter(r)
3  print(ir == r, ir is r)
4
5  iter_lst = iter([-1, 1, -2, 3, 4])
6  iter_lst1 = iter(iter_lst)
7  print(iter_lst is iter_lst1)
```

```
False False
True
```

- The terms “iterable” and “iterator” are sometimes used interchangeably to refer to an object that supports iteration in general
- For clarity, this book has a very strong preference for using the term iterable to refer to an object that supports the iter call, and iterator to refer to an object returned by an iterable on iter that supports the next(I) call
- Range is iterable but not an iterator
- 迭代器的迭代器是自己

--Learning Python, p. 416

Create an Iterator

仅供了解

- To create an object/class as an iterator: implement the methods __iter__() and __next__() to your object
  - The __iter__() method acts similar, you can do operations (initializing etc.), but must always return the iterator object itself （初始化迭代器）
  - The __next__() method also allows you to do operations, and must return the next item in the sequence.
  - To prevent the iteration to go on forever, we can use the **StopIteration statement**

```python
class MyNumbers:
    def __init__(self, a=0):
        self.a = a

    def __iter__(self):
        self.a = 1
        print("__iter__")
        return self

    def __next__(self):
        if self.a <= 7:
            print("next")
            x = self.a
            self.a += 1
            return x
        else:
            raise StopIteration


myclass = MyNumbers()

for x in myclass:
    print(x)
```

```
__iter__
next
1
next
2
next
3
next
4
next
5
next
6
next
7
```

```python
class MyNumbers:
    def __init__(self, a=0):
        self.a = a

    # def __iter__(self):
    #   self.a = 1
    #   print("__iter__")
    #   return self

    # def __next__(self):
    #   if self.a <= 7:
    #       print("next")
    #       x = self.a
    #       self.a += 1
    #       return x
    #   else:
    #       raise StopIteration


myclass = MyNumbers()

for x in myclass:
    print(x)
```

```
TypeError: 'MyNumbers' object is not iterable
```

For 的过程，就是背后调用 __iter__，__next__的过程

Create an Iterator

仅供了解

```python
class MyNumbers:
    def __init__(self, a=0):
        self.a = a

    def __iter__(self):
        self.a = 1
        print("__iter__")
        return self

    def __next__(self):
        if self.a <= 7:
            print("next")
            x = self.a
            self.a += 1
            return x
        else:
            raise StopIteration


myclass = MyNumbers()
myiter = iter(myclass)

for x in myiter:
    print(x)
```

```
__iter__
__iter__
next
1
next
2
next
3
next
4
next
5
next
6
next
7
```

The for loop listens for StopIteration explicitly

The purpose of the for statement is to loop over the sequence provided by an iterator and the exception is used to signal that the iterator is now done; for doesn't catch other exceptions raised by the object being iterated over, just that one

- 系统自定义的list, tuple等实现了__iter__(), __next__()
- 自定义的类型，必须自己实现
- For 遍历过程中，自动调用__iter__(), __next__()的
- 对于定义了__iter__(), __next__()的自定义类，可以通过定义iter(),生成迭代器， next()进行迭代
- iter(), next()分别对应于__iter__(), __next__()

iterable

The assumption behind some Python built-ins is iterable: sum(), =, join(), etc.

- 两个类型可以互相转换的一个必要条件是它们都是iterable
  - list → set
  - a, b, c = x前提是x能够iterable
- find()
- index()
- merge in list, dict
- dict.update()
- str.join()
- file is iterable
- range(): iterable, not iterator

```
1  x = {"1": -1, 2: -2, (1, 2): -3}
2  a, b, c = x
3  print(a, b, c)
```

```
1 2 (1, 2)
```

```
7  i = 0
8  L = [1, 2, 3]
9  i, L[i] = L[i], i
10 print(i, L)
```

A. 1 [0, 2, 3] B. 1 [1, 0, 3]
C. 1 [1, 2, 3] D. 1 [1, 1, 3]

remove to see the answer

itertools

Iterator	Arguments	Results
product()	p, q, … [repeat=1]	cartesian product, equivalent to a nested for-loop
permutations()	p[, r]	r-length tuples, all possible orderings, no repeated elements
combinations()	p, r	r-length tuples, in sorted order, no repeated elements
combinations_with_replacement()	p, r	r-length tuples, in sorted order, with repeated elements

Functions creating iterators for efficient looping

- This module implements a number of iterator building blocks inspired by constructs from APL, Haskell, and SML. Each has been recast in a form suitable for Python
- The module standardizes a core set of fast, memory efficient tools that are useful by themselves or in combination. Together, they form an “iterator algebra” making it possible to construct specialized tools succinctly and efficiently in pure Python
- For instance, SML provides a tabulation tool: tabulate(f) which produces a sequence f(0), f(1), ... The same effect can be achieved in Python by combining map() and count() to form map(f, count())

https://docs.python.org/3/library/itertools.html

collections — Container datatypes

- This module implements specialized container datatypes providing alternatives to Python's general purpose built-in containers, dict, list, set, and tuple.

`namedtuple()`	factory function for creating tuple subclasses with named fields
`deque`	list-like container with fast appends and pops on either end
`ChainMap`	dict-like class for creating a single view of multiple mappings
`Counter`	dict subclass for counting hashable objects
`OrderedDict`	dict subclass that remembers the order entries were added
`defaultdict`	dict subclass that calls a factory function to supply missing values
`UserDict`	wrapper around dictionary objects for easier dict subclassing
`UserList`	wrapper around list objects for easier list subclassing
`UserString`	wrapper around string objects for easier string subclassing

https://docs.python.org/3/library/collections.html

*args, **kwargs

From Lec. 4

*args: Arbitrary positional arguments. **kwargs: Arbitrary keyword arguments

- For arbitrary positional argument, an asterisk (*) is placed before a parameter in function definition which can hold non-keyword variable-length arguments. (Tuple)
- For arbitrary positional argument, a double asterisk (**) is placed before a parameter in a function which can hold keyword variable-length arguments. (Dict)

```
1  def f(*args):
2      for x in args:
3          print(x)
4
5
6  f(1, 2, 3)
7  f(3, 4)
8  f(7)
```

```
1
2
3
3
4
7
```

```
1  def m(*args):
2      z = 1
3      for x in args:
4          z *= x
5
6      print(z)
7
8
9  m(1, 2, 3)
10 m(-1, -2)
11 m(1, 1j, 2j)
```

```
6
2
(-2+0j)
```

```
1  def test_kwargs(**kwargs):
2      for x in kwargs:
3          print(kwargs[x], end=' ')
4
5      print()
6
7
8  test_kwargs(x=1)
9  test_kwargs(x=1, y=2)
10 test_kwargs(x=1, y=2, z=3)
11 test_kwargs(x=1, y=2, z=3, u=4)
```

```
1
1 2
1 2 3
1 2 3 4
```

*: unpack iterable

- starred assignment target must be in a list or tuple

```python
def test_star(it):
    x, *f = it
    print(x, f)


test_star([_ for _ in range(7)])
test_star({1: -1, 2: 2, 3: 3})

lst = [_ for _ in range(7)]
# *f = lst  # SyntaxError: starred assignment target must be in a list or tuple
(*f,) = lst


def f(a, b, c):
    return a + b * c


dl = [1, 2, 3]
print(f(*dl))
print(*dl, dl)

sg = [x for x in range(10)]
print(*sg)
```

```
0 [1, 2, 3, 4, 5, 6]
1 [2, 3]
7
1 2 3 [1, 2, 3]
0 1 2 3 4 5 6 7 8 9
```

The same for **.
Recall how to merge two dicts:
c = {**a, **b}

all(), any()

- all(iterable)
  - Return True if all elements of the iterable are true (or if the iterable is empty).
  - $x_1 \land x_2 \land \cdots \land x_n$
- any(iterable)
  - Return True if any element of the iterable is true. If the iterable is empty, return False.
  - $x_1 \lor x_2 \lor \cdots \lor x_n$

```
1  assert any([False, False, True, False]) == True
2  assert all([True, True, True, False]) == False
```

Lec. 4: Literal patterns (string literals, number literals, True, False, and None)

Structural Pattern Matching (3.10)

- match subject:
  case <pattern_1>:
    <action_1>
  case <pattern_2>:
    <action_2>
  case <pattern_3>:
    <action_3>
  case _:
    <action_wildcard>
- 从上往下对比，如果遇到符合的pattern，就执行该case的代码，结束后离开match
- _ : 表示wildcard，会match任意的情况，可以省略
- 比C/C++中switch要强很多

```python
if x == 1:
    print("Monday")
elif x == 2:
    print("Tuesday")
elif x == 3:
    print("Wednesday")
elif x == 4:
    print("Thursday")
elif x == 5:
    print("Friday")
elif x == 6:
    print("Saturday")
else:
    print("Sunday")
```

```python
def test_day(x):
    match x:
        case 1:
            print("Monday")
        case 2:
            print("Tuesday")
        case 3:
            print("Wednesday")
        case 4:
            print("Thursday")
        case 5:
            print("Friday")
        case 6:
            print("Saturday")
        case _:  # try to remove it.
            print("Sunday")


for x in range(1, 10):
    test_day(x)
```

```
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Sunday
Sunday
Sunday
```

结构更清晰，可读性更好

Structural Pattern Matching

- As an example to motivate this tutorial, you will be writing a text adventure.
- That is a form of interactive fiction where the user enters text commands to interact with a fictional world and receives text descriptions of what happens.
- Commands will be simplified forms of natural language like get sword, attack dragon, go north, enter shop or buy cheese.

```
command = input("What are you doing next? ")
[action, obj] = command.split()
```

- The problem with that line of code is that it's missing something: what if the user types more or fewer than 2 words?
- To prevent this problem you can either check the length of the list of words, or capture the ValueError that the statement above would raise.
- Structural Pattern Matching

https://peps.python.org/pep-0636/

Matching sequences

```
match command.split():
    case [action, obj]:
        ... # interpret action, obj
```

- The match statement evaluates the “subject” (the value after the match keyword), and checks it against the pattern (the code next to case). A pattern is able to do two different things:
- Verify that the subject has certain structure. In your case, the [action, obj] pattern matches any sequence of exactly two elements. This is called matching
- It will bind some names in the pattern to component elements of your subject. In this case, if the list has two elements, it will bind action = subject[0] and obj = subject[1].
- If there’s a match, the statements inside the case block will be executed with the bound variables. If there’s no match, nothing happens and the statement after match is executed next.

Matching multiple patterns

- Even if most commands have the action/object form, you might want to have user commands of different lengths. For example, you might want to add single verbs with no object like look or quit. A match statement can (and is likely to) have more than one case:

```
match command.split():
  case [action]:
    ... # interpret single-verb action
  case [action, obj]:
    ... # interpret action, obj
```

- The match statement will check patterns from top to bottom. If the pattern doesn't match the subject, the next pattern will be tried. However, once the first matching pattern is found, the body of that case is executed, and all further cases are ignored. This is similar to the way that an if/elif/elif/... statement works.

Matching specific values

- Your code still needs to look at the specific actions and conditionally execute different logic depending on the specific action (e.g., quit, attack, or buy). You could do that using a chain of if/elif/elif/..., or using a dictionary of functions, but here we'll leverage pattern matching to solve that task. Instead of a variable, you can use literal values in patterns (like "quit", 42, or None). This allows you to write:

```
match command.split():
    case ["quit"]:
        print("Goodbye!")
        quit_game()
    case ["look"]:
        current_room.describe()
    case ["get", obj]:
        character.get(obj, current_room)
    case ["go", direction]:
        current_room = current_room.neighbor(direction)
    # The rest of your commands go here
```

- A pattern like ["get", obj] will match only 2-element sequences that have a first element equal to "get". It will also bind obj = subject[1].
- As you can see in the go case, we also can use different variable names in different patterns.
- Literal values are compared with the == operator except for the constants True, False and None which are compared with the is operator.

Matching multiple values

- A player may be able to drop multiple items by using a series of commands drop key, drop sword, drop cheese. This interface might be cumbersome, and you might like to allow dropping multiple items in a single command, like drop key sword cheese. In this case you don't know beforehand how many words will be in the command, but you can use extended unpacking in patterns in the same way that they are allowed in assignments:

```
match command.split():
    case ["drop", *objects]:
        for obj in objects:
            character.drop(obj, current_room)
    # The rest of your commands go here
```

- This will match any sequences having "drop" as its first elements. All remaining elements will be captured in a list object which will be bound to the objects variable.
- This syntax has similar restrictions as sequence unpacking: you can not have more than one starred name in a pattern.

Composing patterns

- This is a good moment to step back from the examples and understand how the patterns that you have been using are built. Patterns can be nested within each other, and we have been doing that implicitly in the examples above.
- There are some “simple” patterns (“simple” here meaning that they do not contain other patterns) that we’ve seen:
- Capture patterns (stand-alone names like direction, action, objects). We never discussed these separately, but used them as part of other patterns.
- Literal patterns (string literals, number literals, True, False, and None)
- The wildcard pattern _
- Until now, the only non-simple pattern we have experimented with is the sequence pattern. Each element in a sequence pattern can in fact be any other pattern. This means that you could write a pattern like ["first", (left, right), _, *rest]. This will match subjects which are a sequence of at least three elements, where the first one is equal to "first" and the second one is in turn a sequence of two elements. It will also bind left=subject[1][0], right=subject[1][1], and rest = subject[3:]

Or patterns

- Going back to the adventure game example, you may find that you'd like to have several patterns resulting in the same outcome. For example, you might want the commands north and go north to be equivalent. You may also desire to have aliases for get X, pick up X and pick X up for any X.
- The | symbol in patterns combines them as alternatives. You could for example write:

```
match command.split():
    ... # Other cases
    case ["north"] | ["go", "north"]:
        current_room =
current_room.neighbor("north")
    case ["get", obj] | ["pick", "up", obj] |
        ["pick", obj, "up"]:
            ... # Code for picking up the given
        object
```

- This is called an or pattern and will produce the expected result. Patterns are tried from left to right; this may be relevant to know what is bound if more than one alternative matches. An important restriction when writing or patterns is that all alternatives should bind the same variables. So a pattern [1, x] | [2, y] is not allowed because it would make unclear which variable would be bound after a successful match. [1, x] | [2, x] is perfectly fine and will always bind x if successful.

Capturing matched sub-patterns

- The first version of our “go” command was written with a ["go", direction] pattern. The change we did in our last version using the pattern ["north"] | ["go", "north"] has some benefits but also some drawbacks in comparison: the latest version allows the alias, but also has the direction hardcoded, which will force us to actually have separate patterns for north/south/east/west. This leads to some code duplication, but at the same time we get better input validation, and we will not be getting into that branch if the command entered by the user is "go figure!" instead of a direction.
- We could try to get the best of both worlds doing the following (I’ll omit the aliased version without “go” for brevity):

```
match command.split():
  case ["go", ("north" | "south" | "east" | "west")]:
      current_room = current_room.neighbor(...)
      # how do I know which direction to go?
```

- This code is a single branch, and it verifies that the word after “go” is really a direction. But the code moving the player around needs to know which one was chosen and has no way to do so. What we need is a pattern that behaves like the or pattern but at the same time does a capture. We can do so with an as pattern:

```
match command.split():
  case ["go", ("north" | "south" | "east" | "west") as direction]:
      current_room = current_room.neighbor(direction)
```

- The as-pattern matches whatever pattern is on its left-hand side, but also binds the value to a name.

Adding conditions to patterns

- The patterns we have explored above can do some powerful data filtering, but sometimes you may wish for the full power of a boolean expression. Let’s say that you would actually like to allow a “go” command only in a restricted set of directions based on the possible exits from the current_room. We can achieve that by adding a guard to our case. Guards consist of the if keyword followed by any expression:

```
match command.split():
    case ["go", direction] if direction in
current_room.exits:
        current_room =
current_room.neighbor(direction)
    case ["go", _]:
        print("Sorry, you can't go that way")
```

- The guard is not part of the pattern, it’s part of the case. It’s only checked if the pattern matches, and after all the pattern variables have been bound (that’s why the condition can use the direction variable in the example above). If the pattern matches and the condition is truthy, the body of the case executes normally. If the pattern matches but the condition is falsy, the match statement proceeds to check the next case as if the pattern hadn’t matched (with the possible side-effect of having already bound some variables).

Structural Pattern Matching

- Matching sequences
- Matching multiple patterns
- Matching specific values
- Matching multiple values
- Adding a wildcard
- Composing patterns
- Or patterns
- Capturing matched sub-patterns
- Adding conditions to patterns

Self-learning

- Adding a UI: Matching objects
- Matching positional attributes
- Matching against constants and enums
- Going to the cloud: Mappings
- Matching builtin classes

https://peps.python.org/pep-0636/

Generator

List comprehension (列表推导)

- Now we will introduce a simple alternative to create a list of squares

List是python的核心

```
squares = []

for x in range(10):
    squares.append(x**2)
print(squares)

squares = [x**2 for x in range(10)]
print(squares)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

- Grammar

单重循环 [expression for target in iterable if conditon]
多重循环 [expression for target1 in iterable1 if condition1
for target2 in iterable2 if condition2 ..
for targetN in iterableN if conditionN]

- A list comprehension consists of brackets [ ] containing an expression followed by a for clause, then zero or more for or if clauses
- The result will be a new list resulting from evaluating the expression in the context of the for and if clauses which follow it
- List comprehension can be nested

- **if**
- **嵌套**
- **函数**

```python
print([2 * x for x in range(6) if x % 2 == 0])
print([(x, y) for x in [1, 2, 3] for y in [3, 1, 4] if x != y])

vec = [-4, -2, 0, 2, 4]
print([x * 2 for x in vec])
print([x for x in vec if x >= 0])
print([abs(x) for x in vec])

fruit = ["  banana", "   loganberry", "passion fruit   "]
print([x.strip() for x in fruit])

print([(x, x**2) for x in range(6)])

from math import pi

print([str(round(pi, i)) for i in range(6)])

matrix = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
print([[row[i] for row in matrix] for i in range(4)])
```

```
[0, 4, 8]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
[-8, -4, 0, 4, 8]
[0, 2, 4]
[4, 2, 0, 2, 4]
['banana', 'loganberry', 'passion fruit']
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
['3.0', '3.1', '3.14', '3.142', '3.1416', '3.14159']
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

如何创建新的list

- ☐ for
- ☐ list comprehension
- ☐ [0]*n
- ☐ list(range()), list(set), list(str)

- ✓ 根据需要合理使用
- ✓ 清晰，可读，正确
- ✓ 比for更快

List is fast

- Python的解释器CPython基于C
- 纯python代码会很慢，部分python代码后面有c实现
- 尽可能用系统自带的实现，譬如sum()
- 能够用列表操作，就不要用for， for很慢. Python对list提供了很多操作
- List slice + List comprehension:
  lst1[a,b,d] = lst2[a,b,d]
  # Sometimes, SIMD
- Single instruction, multiple data (SIMD) is a type of parallel processing. (单指令多数据: 一条高速路，多条车道)

```
4766  import time
4767
4768  print("Test by slice")
4769  n = 1000
4770  lst1 = [x for x in range(10**6)]
4771
4772  time_begin = time.time()
4773  for _ in range(n):
4774      lst1[0::2] = [1]*(10**6//2)
4775  time_end = time.time()
4776
4777  print((time_end-time_begin)/n)
4778
4779  print("Test by Loop")
4780  n = 1000
4781  lst2 = [x for x in range(10**6)]
4782
4783  time_begin = time.time()
4784  for _ in range(n):
4785      for i in range(10**6//2):
4786          lst2[2*i] = 1
4787  time_end = time.time()
4788
4789  print((time_end-time_begin)/n)
4790  print(lst1 == lst2)
```

```
Test by slice
0.00299402117729187
Test by Loop
0.024240193128585816
True
```

大概相差一个数量级

generator（生成器、发电机）

- Why range(n)? Why not list(n)?
  - list()=[]*(10**20)？天文数字的空间存储数据！
  - range(n): n =10**20
  - We need only one element in each iteration!
    - 没有必要一次性生成所有元素。要的时候再生成（按需供应）
- Generator: Produce results on demand
  - Generator functions (available since 2.3) are coded as normal def statements, but use yield statements to return results one at a time, suspending and resuming their state between each
  - Generator expressions (available since 2.4) are similar to the list comprehensions of the prior section, but they return an object that produces results on demand instead of building a result list
- Because neither constructs a result list all at once, they save memory space and allow computation time to be split across result requests

yield：generator function

```
def gensquares(N):
    for i in range(N):
        yield i ** 2 # Resume here later
```

- This function yields a value, and so returns to its caller, each time through the loop; when it is resumed, its prior state is restored, including the last values of its variables i and N, and control picks up again immediately after the yield statement
- 除了返回函数值，yield比return还多了中间状态，可以循环返回
- Generator是自己的iterator

```
1  def gensquares(N):
2      for i in range(N):
3          yield i**2  # Resume here later
4
5
6  for i in gensquares(5):  # range()
7      print(i, end=" ")
```

```
0 1 4 9 16
```

```
1  x = gensquares(4)
2  ix = iter(x)
3  print(ix == x, ix is x)
4  print(x)
```

```
True True
<generator object gensquares at 0x112417920>
```

```
1  print(next(x))
2  print(next(x))
3  print(next(x))
4  print(next(x))
5  # print(next(x)) # error: last element
```

```
0
1
4
9
```

Generator Expressions: Iterables + Comprehensions

Syntactically, generator expressions are just like normal list comprehensions, and support all their syntax — including if filters and loop nesting—but they are enclosed in parentheses instead of square brackets (like tuples, their enclosing parentheses are often optional)

```
1   lst = [x ** 2 for x in range(4)]
2   print(type(lst))
3
4   ge = (x ** 2 for x in range(4))
5   print(type(ge))
6
7   lst1 = list(x ** 2 for x in range(4))
8   print(type(lst1))
9
10  ge1 = x ** 2 for x in range(4) #error
11  print(type(ge))
```

```
<class 'list'>
<class 'generator'>
<class 'list'>
```

不加括号()[]是语法错误

generator→list/dict/set/tuple comprehension

- tuple(generator): 元组推导
- dict(generator): 字典推导
- set(geneartor): 集合推导

```
g = ( 2*x+1 for x in range(7))
lg = tuple(g)
print(type(g), type(lg), lg)

g = ( 2*x+1 for x in range(7))
lg = set(g)
print(type(g), type(lg), lg)

# g = (2*x+1:x for x in range(7)) # error
lg = { 2*x+1:x for x in range(7)}
print(type(g), type(lg), lg)
```

```
<class 'generator'> <class 'tuple'> (1, 3, 5, 7, 9, 11, 13)
<class 'generator'> <class 'set'> {1, 3, 5, 7, 9, 11, 13}
<class 'generator'> <class 'dict'> {1: 0, 3: 1, 5: 2, 7: 3, 9: 4, 11: 5, 13: 6}
```

```
for num in (x ** 2 for x in range(4)): # Calls next() automatically
    print('%s, %s' % (num, num / 2.0))

print(''.join(x.upper() for x in 'aaa,bbb,ccc'.split(',')))

a, b, c = (x + '\n' for x in 'aaa,bbb,ccc'.split(','))
print(a, b, c)

print(sum(x ** 2 for x in range(4)))
print(sorted(x ** 2-1.5*x for x in range(4)))
print(sorted((x ** 2-1.5*x for x in range(4)), reverse=True))
```

```
0, 0.0
1, 0.5
4, 2.0
9, 4.5
AAABBBCCC
aaa
 bbb
 ccc

14
[-0.5, 0.0, 1.0, 4.5]
[4.5, 1.0, 0.0, -0.5]
```

```
def timesfour(S): # Generator function
    for c in S:
        yield c * 4

G = timesfour('spam')
print(list(G))
```

```
['ssss', 'pppp', 'aaaa', 'mmmm']
```

```
line = 'aa bbb c'
print(''.join(x.upper() for x in line.split() if len(x) > 1)) # Expression

def gensub(line): # Function
    for x in line.split():
        if len(x) > 1:
            yield x.upper()
print(''.join(gensub(line)))
```

```
AABBB
AABBB
```

list(generator)

- list(generator)会在内部将generator循环一遍，即generator的iter()到达末尾

```
sg = (x for x in "hello world. 苟利")
print(next(sg))

lis = list(sg)
print(lis)

print(next(sg))
```

很隐秘的bug

```
h
['e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd', '.', ' ', '苟', '利']
Traceback (most recent call last):
  File "c:/Users/popeC/OneDrive/CS124计算导论/2020 秋季/lecture notes/course_code.py", line 1465, in <module>
    print(next(sg))
StopIteration
```

```
sg = (x for x in "hello world. 苟利")
print(type(sg))
lis = list(sg)
print(lis)
print(next(sg))
```

```
<class 'generator'>
['h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd', '.', ' ', '苟', '利']
Traceback (most recent call last):
  File "c:/Users/popeC/OneDrive/CS124计算导论/2020 秋季/lecture notes/course_code.py", line 1471, in <module>
    print(next(sg))
StopIteration
```

- 不要一遍修改一个iterator，一边遍历它：bug相对论

First Class Citizen

Function: first-class citizen in Python

- What is a function? def ……
- A first-class function is not a particular kind of function. All functions in Python are first-class functions.
- FCC: Treat function as int, str, float, etc.
  - Can be used as parameters
  - Can be used as a return value
  - Can be assigned to variables
  - Can be stored in data structures such as hash tables, lists, ...
  - Actually, very roughly and simply put, FCF's are variables of the type 'function' (or variables which point to a function). You can do with them everything you can do with a 'normal' variable
- To say that functions are first-class in a certain programming language means that they can be passed around and manipulated similarly to how you would pass around and manipulate other kinds of objects (like integers or strings)
  - You can assign a function to a variable, pass it as an argument to another function, etc
  - The distinction is not that individual functions can be first class or not, but that entire languages may treat functions as first-class objects, or may not

Functions are data

- Python built-in functions: len, id, type, str, int, float, print

函数式编程：数据不变，函数变

```
# function assignment
mytype = type
myid = id
mylen = len
myprint  = print
poem = "苟利国家生死以，岂因祸福避趋之"
print(poem, type(poem), id(poem), len(poem))
myprint(poem, mytype(poem), myid(poem), mylen(poem))

# functions are parameters
def func_type(func):
    return type(func)

def add(x, y):
    return x + y

print(func_type(id))
print(func_type(len))
print(func_type(myprint))
print(func_type(myid))
print(func_type(func_type))
print(func_type(add))
```

```
# functions are data
actions = [type, str, id, len]

for act in actions:
    print(act("不讲武德，耗子尾汁"))

# return value
def make(N):
    def action(x):
        return x ** N

    return action

f1 = make(2)
f2 = make(3)
f3 = make(4)

print(f1(3), f2(4), f3(5))
```

```
苟利国家生死以，岂因祸福避趋之 <class 'str'> 1818868423456 15
苟利国家生死以，岂因祸福避趋之 <class 'str'> 1818868423456 15
<class 'builtin_function_or_method'>
<class 'builtin_function_or_method'>
<class 'builtin_function_or_method'>
<class 'builtin_function_or_method'>
<class 'function'>
<class 'function'>
```

```
<class 'str'>
不讲武德，耗子尾汁
1818868861616
9
9 64 625
```

callable(), __call__()

- Everything in python is an object
- Function is callable(): __call__()

```python
def greeting():
    print("hello, SJTU -- TOP3")


greeting()

greeting.__call__()

print(callable(greeting))
print(callable(1))
print(callable("hello world"))
```

```
hello, SJTU -- TOP3
hello, SJTU -- TOP3
True
False
False
```

```python
class Test:
    pass


class TestCall:
    def __call__(self, x):
        return x + 100


t = Test()
tc = TestCall()

print(callable(t))
print(callable(tc))
print(tc(2023))
```

```
False
True
2123
```

Default arguments

- 默认参数可以看做函数的属性
- 在定义的时候初始化

- 函数内的变量会随着函数的调用完成而销毁
- 默认参数是函数的一种状态，在函数定义时创建，会一直保存下来
- 默认参数在参数列表中要放在非默认参数后面
- 参数名可以和变量名重名
- 一般情况下，默认参数作为系统的属性，不要修改
- 如果需要可用None作为默认参数

```
1024
93174.3733866435
```

```
[1]
[1, 2]
[1, 2, 3]
[1, 2, 3, 4]
[1, 2, 3, 4, 5]
[1]
[2]
[3]
[4]
[5]
```

```python
x = 10


def powx(a, x=x):
    return a**x


print(powx(2))
print(powx(3.14))
```

```python
def test_default(x, lst=[]):
    lst.append(x)
    return lst


print(test_default(1))
print(test_default(2))
print(test_default(3))
print(test_default(4))
print(test_default(5))
```

```python
def test_default(x, lst=None):
    if not lst:
        lst = []

    lst.append(x)
    return lst


print(test_default(1))
print(test_default(2))
print(test_default(3))
print(test_default(4))
print(test_default(5))
```

Default arguments

- 函数的默认参数保存在funcname.__defaults__中
- Default parameter values are evaluated from left to right when the function definition is executed
  - This means that the expression is evaluated once, when the function is defined, and that the same “pre-computed” value is used for each call
  - This is especially important to understand when a default parameter is a mutable object, such as a list or a dictionary: if the function modifies the object (e.g. by appending an item to a list), the default value is in effect modified.
  - This is generally not what was intended. A way around this is to use None as the default, and explicitly test for it in the body of the function
- 默认参数在函数定义时(注意：不是执行时)初始化，id(x)永远不变
  - 如果默认参数的类型可以被修改，那么默认参数可以变
  - 如果不可以被修改，那么不变
  - 无论如何，id不变

- 默认参数可以看做函数的属性
- 在定义的时候初始化

https://docs.python.org/3.5/reference/compound_stmts.html#function-definitions

- 设置为None或者一个不会出现的值
- None is immutable

```python
def test_default(x, lst=[]):
    print(lst, type(lst), id(lst))
    lst.append(x)
    return lst


print(test_default.__defaults__[0])
print(test_default(1))
print(test_default(2))
print(test_default(3))
print(test_default(4))
print(test_default(5))
print(test_default.__defaults__[0])
```

```
[]
[] <class 'list'> 140425493656640
[1]
[1] <class 'list'> 140425493656640
[1, 2]
[1, 2] <class 'list'> 140425493656640
[1, 2, 3]
[1, 2, 3] <class 'list'> 140425493656640
[1, 2, 3, 4]
[1, 2, 3, 4] <class 'list'> 140425493656640
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5]
```

```python
def test_default(x, lst=None):
    print(lst, type(lst), id(lst))
    if not lst:
        lst = []

    lst.append(x)
    return lst


print(test_default.__defaults__[0])
print(test_default(1))
print(test_default(2))
print(test_default(3))
print(test_default(4))
print(test_default(5))
print(test_default.__defaults__[0])
```

```
None
None <class 'NoneType'> 4332715632
[1]
None <class 'NoneType'> 4332715632
[2]
None <class 'NoneType'> 4332715632
[3]
None <class 'NoneType'> 4332715632
[4]
None <class 'NoneType'> 4332715632
[5]
None
```

```python
def test_default(x, const=-1):
    print(f"Before {x = }, {const = }")
    const += x
    print(f"After {x = }, { const = }")


test_default(x=0)
test_default(x=1)
test_default(x=2)
test_default(x=3)
test_default(x=-1)
```

```
Before x = 0, const = -1
After x = 0,  const = -1
Before x = 1, const = -1
After x = 1,  const = 0
Before x = 2, const = -1
After x = 2,  const = 1
Before x = 3, const = -1
After x = 3,  const = 2
Before x = -1, const = -1
After x = -1,  const = -2
```

const每次都会reset

Function Attributes

- dir(func): func内部所有的属性
- __name__
- __code__
- __doc__
- __dict__
- Self defined attributes: func.variable

```
['__annotations__', '__call__', '__class__', '__closure__', '__code__',
'__defaults__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__get__', '__getattribute__', '__globals__',
'__gt__', '__hash__', '__init__', '__init_subclass__', '__kwdefaults__',
'__le__', '__lt__', '__module__', '__name__', '__ne__', '__new__',
'__qualname__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', 'greeting', 'time']
hi 2020
func
<code object func at 0x7fd3ef5972f0, file "/Users/fancheng/OneDrive/
CS124计算导论/2020 秋季/lecture notes/course_code.py", line 3596>

    Life is short. Use python

{'greeting': 'hi', 'time': '2020'}
8784264391527
```

```
def func():
    '''
    Life is short. Use python
    '''
    def say_hello():
        print("Hello Python")

func.greeting = "hi"
func.time = "2020"

print(dir(func))
print(func.greeting, func.time)
print(func.__name__)
print(func.__code__)
print(func.__doc__)
print(func.__dict__)
print(func.__hash__())
```

Function: late binding

- 函数如果使用外部变量，那么外部变量的值在函数调用时才确定，不是在函数定义时决定
  - 函数在定义的时候不会自动执行
  - 函数的参数在调用时才会确定(late)

```
# late binding: 调用的时候才确定a的值
a =  "耗子尾汁"
def powa(x):
    return x**a

a = 10
print(powa(2))
print(powa(1.6))

a = 2.0
print(powa(2))
print(powa(1.6))

a = -1
print(powa(2))
print(powa(1.6))
```

```
10
1024
109.95116277760006
4.0
2.5600000000000005
0.5
0.625
```

去掉a=“耗子尾汁”也不影响
powa由调用时的a决定，不是定义时的a

Late binding: solution

- 函数如果使用外部变量，那么外部变量的值在函数调用时才确定，不是在函数定义时决定
- 如果需要在定义时就确定：用默认参数（默认参数在函数定义时就确定了）

```
# late binding: 调用的时候才确定a的值

a = 7
def powa(x, a = a):
    return x**a

print(powa(2))
print(powa(1.6))

a = 10
print(powa(2))
print(powa(1.6))

a = 2.0
print(powa(2))
print(powa(1.6))

a = -1
print(powa(2))
print(powa(1.6))
```

```
128
26.84354560000001
128
26.84354560000001
128
26.84354560000001
128
26.84354560000001
```

powa中a=7
即外部变量⇒内部变量

- 函数在定义的时候不会自动执行
- 函数的参数在调用时才会确定
- 默认参数在函数定义时就确定了

循环可以展开

```
def pow_x():
    plist = []

    for i in range(5):
        def powa(x):
            return x**i

        plist.append(powa)

    return plist

for i in range(5):
    print(pow_x()[i](2))
```

```
16
16
16
16
16
```

```
def pow_x():
    plist = []

    for i in range(5):
        def powa(x, i=i):
            return x**i

        plist.append(powa)

    return plist

for i in range(5):
    print(pow_x()[i](2))
```

```
1
2
4
8
16
```

三个要点

- 函数在定义的时候不会自动执行
- 函数的参数在调用时才会确定
- 默认参数在函数定义时就确定了

Closure 闭包

- Functions could be nested inside another function
- Closure is a functional programming concept, which is called function factory (函数工厂) in design pattern. It should satisfy the following:
  1. 函数嵌套：内部函数、外部函数
  2. 内部函数引用外部函数的数据
  3. 内部函数被外部函数返回
- __closure__: 闭包数据

```python
# # return value
def make(N):
    def action(x):
        return x ** N

    return action

f1 = make(2)
f2 = make(3)
f3 = make(4)

print(f1(3), f2(4), f3(5))
```

```
9 64 625
```

```python
def outer(a):

  def inner(b):
    return a + b

  return inner

test = outer(10)
print(test(2), test(3))
print(outer(-1)(-20))
print(test.__closure__)
print(test.__closure__[0].cell_contents)
```

```
12 13
-21
(<cell at 0x7fa5aa32ed00: int object at 0x10c98abc0>,)
10
```

Decorator 装饰器

- Decorator are very powerful and useful tool in Python since it allows programmers to modify the behavior of function or class. Decorators allow us to wrap another function in order to extend the behavior of wrapped function, without permanently modifying it
- In Decorators, functions are taken as the argument into another function and then called inside the wrapper function

- 函数闭包：被装饰函数作为参数传入
- 内嵌函数调用被装饰函数，同时增加新的行为
- 外部函数返回内嵌函数

不改变被装饰函数代码的情况下，增广其功能

- Learning python: Ch. 39

Decorator: @

- 等价写法 @decorator_name
- @decorator_name 放在被装饰函数定义前
- 回顾：类方法 @classmethod

```
# defining a decorator
def hello_decorator(func):

    # inner1 is a Wrapper function in
    # which the argument is called

    # inner function can access the outer local
    # functions like in this case "func"
    def inner1():
        print("Hello, this is before function execution")

        # calling the actual function now
        # inside the wrapper function.
        func()

        print("This is after function execution")

    return inner1


# defining a function, to be called inside wrapper
def function_to_be_used():
    print("This is inside the function !!")


# passing 'function_to_be_used' inside the
# decorator to control its behavior
function_to_be_used = hello_decorator(function_to_be_used)


# calling the function
function_to_be_used()
```

```
Hello, this is before function execution
This is inside the function !!
This is after function execution
```

```
def hello_decorator(func):

    def inner1():
        print("Hello, this is before function execution")
        func()
        print("This is after function execution")


    return inner1


@hello_decorator
def function_to_be_used():
    print("This is inside the function !!")


function_to_be_used()

@hello_decorator
def function_to_be_used_2():
    print("耗子尾汁")


function_to_be_used_2()
```

```
Hello, this is before function execution
This is inside the function !!
This is after function execution
Hello, this is before function execution
耗子尾汁
This is after function execution
```

Decorate 3rd party functions

- It is also possible to decorate third party functions, e.g. functions we import from a module. We can't use the Python syntax with the "at" sign in this case

```
from math import sin, cos

def our_decorator(func):
    def function_wrapper(x):
        print("Before calling " + func.__name__)
        res = func(x)
        print(res)
        print("After calling " + func.__name__)
    return function_wrapper

sin = our_decorator(sin)
cos = our_decorator(cos)

for f in [sin, cos]:
    f(3.1415)
```

```
Before calling sin
9.265358966049024e-05
After calling sin
Before calling cos
-0.9999999957076562
After calling cos
```

Functions with arbitrary parameters

- f(*args, **kwargs)

```
from random import random, randint, choice

def our_decorator(func):
    def function_wrapper(*args, **kwargs):
        print("Before calling " + func.__name__)
        res = func(*args, **kwargs)
        print(res)
        print("After calling " + func.__name__)
    return function_wrapper

random = our_decorator(random)
randint = our_decorator(randint)
choice = our_decorator(choice)

random()
randint(3, 8)
choice([4, 5, 6])
```

```
Before calling random
0.08416535367526257
After calling random
Before calling randint
3
After calling randint
Before calling choice
4
After calling choice
```

Functions with return value

```python
def hello_decorator(func):
    def inner1(*args, **kwargs):

        print("before Execution")

        # getting the returned value
        returned_value = func(*args, **kwargs)
        print("after Execution")

        # returning the value to the original frame
        return returned_value

    return inner1


# adding decorator to the function
@hello_decorator
def sum_two_numbers(a, b):
    print("Inside the function")
    return a + b

a, b = 1, 2

# getting the value through return of the function
print("Sum =", sum_two_numbers(a, b))
```

```
before Execution
Inside the function
after Execution
Sum = 3
```

https://www.geeksforgeeks.org/decorators-in-python/

```python
def argument_test_natural_number(f):
    def helper(x):
        if type(x) == int and x > 0:
            return f(x)
        else:
            raise Exception("Argument is not an integer")
    return helper

@argument_test_natural_number
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)

for i in range(1,10):
    print(i, factorial(i))

print(factorial(-1))
```

```
1 1
2 2
3 6
4 24
5 120
6 720
7 5040
8 40320
9 362880
Traceback (most recent call last):
  File "/Users/fancheng/OneDrive/CS124计算导论/2020 秋季/lecture notes/
  course_code.py", line 3719, in <module>
    print(factorial(-1))
  File "/Users/fancheng/OneDrive/CS124计算导论/2020 秋季/lecture notes/
  course_code.py", line 3706, in helper
    raise Exception("Argument is not an integer")
Exception: Argument is not an integer
```

```python
def call_counter(func):
    def helper(*args, **kwargs):
        helper.calls += 1
        return func(*args, **kwargs)
    helper.calls = 0

    return helper

@call_counter
def succ(x):
    return x + 1

@call_counter
def mul1(x, y=1):
    return x*y + 1

print(succ.calls)
for i in range(10):
    succ(i)
mul1(3, 4)
mul1(4)
mul1(y=3, x=2)

print(succ.calls)
print(mul1.calls)
```

```
0
10
3
```

https://www.python-course.eu/python3_decorators.php#First-Steps-to-Decorators

```python
import time
import math


def calculate_time(func):

    # added arguments inside the inner1,
    # if function takes any arguments,
    # can be added like this.
    def inner1(*args, **kwargs):

        # storing time before function execution
        begin = time.time()

        r = func(*args, **kwargs)

        # storing time after function execution
        end = time.time()
        print("Total time taken in : ", func.__name__, end - begin)

    return inner1
```

```python
@calculate_time
def factorial(num):

    # sleep 2 seconds because it takes very less time
    # so that you can see the actual difference
    time.sleep(2)
    print(math.factorial(num))

# calling the function.
factorial(10)
```

```
3628800
Total time taken in :  factorial 2.00455379486084
```

@property

```python
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y


pt = Point(12, 5)
print(pt.x, pt.y)

pt.x, pt.y = 42, 6
print(pt.x, pt.y)
```

- Python语言灵活性，可以直接修改属性
- 如何修改属性的参数，不改变属性名字
- property()
- @property

```python
class Circle:
    def __init__(self, radius):
        self._radius = radius

    def _get_radius(self):
        print("Get radius")
        return self._radius

    def _set_radius(self, value):
        print("Set radius")
        self._radius = value

    def _del_radius(self):
        print("Delete radius")
        del self._radius

    radius = property(
        fget=_get_radius,
        fset=_set_radius,
        fdel=_del_radius,
        doc="The radius property."
    )
c = Circle(7)
print(c.radius)

c.radius = 10
print(c.radius)

del c.radius
```

```python
class Circle:
    def __init__(self, radius):
        self._radius = radius

    @property
    def radius(self):
        """The radius property."""
        print("Get radius")
        return self._radius

    @radius.setter
    def radius(self, value):
        print("Set radius")
        self._radius = value

    @radius.deleter
    def radius(self):
        print("Delete radius")
        del self._radius

c = Circle(7)
print(c.radius)

c.radius = 10
print(c.radius)

del c.radius
```

Pythonic

- 论一个python程序员的修养
- 最高原则 import this
  - KISS: Keep It Simple
  - EIBTI: Explicit is better than implicit
- Python的风格
  - List comprehension, generator, decorator
- Dataclasses – @dataclass
  - https://docs.python.org/3/library/dataclasses.html