Module 01 – Advanced techniques

Agenda

- Lambda functions
- Filter and map
- is and id
- Decorators
- Iterators and itertools
- Generators
- Garbage collector
- Random

Lambda

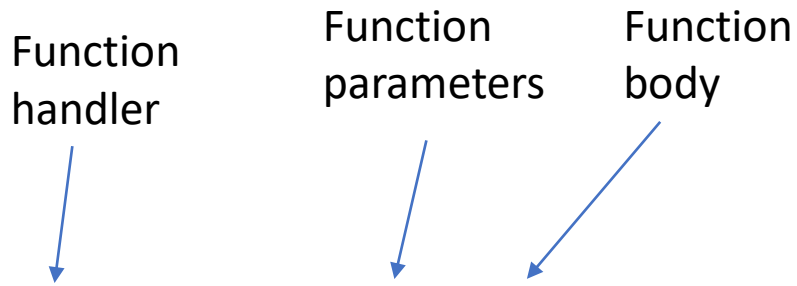
Python supports the creation of anonymous functions (i.e. functions that are not bound to a name) at runtime, using a construct caked “lambda”

This is a very powerful concept that well integrated into Python

The most common use of lambda functions is as callback

Lambda definition

```
def f(x):
    return x**2
print(f(8)) # print 64
```




```
g= lambda x: x**2

print(g(8)) #print 64
```

Filter function

- **resultiter =filter(function, iterable)**
- filter function returns an iterator of elements of iterable for which function returns true.
- iterable may be either a sequence or an iterator
- The filter iterates through all elements of iterable, sends them(one by one) to function and includes in resultiter only elements for which the function returns True
- If the function is not None, elements from input iterable will be include in a result only if them evaluates as True

Filter function

Demo

filter function — demo

- For example:

```
list= [2, 18, 9, 22, 17, 24, 8, 12, 27]
seq = filter(lambda x: x % 3 == 0, list)
for val in seq:
    print(val)                    #18,9, 24,12,27

seq = filter(lambda x: x < 0, range(-5,5))
for val in seq:
    print(val)                    #-5, -4, -3, -2,-1
```

map function

- **resultIter = map(function, iterable, [iterables])**
- Return an iterator that applies function to every item of iterable.
- * If additional iterables arguments are passed, function must take that many arguments and is applied to the items from all iterables corresponding
  - — If the number of items in all iterables is not even, the map function y will only iterate through the common items, in python 3 (the number of iterations is defined by minimal sequence length) F
    - In python 2, the map function iterates through all items and sends | None for absent correspondings (the number of iterations is defined by maximal sequence length

Map function

Demo

map function – cont'd

For Example:

```
list = [2, 18, 9, 22, 17, 24, 8, 12, 27]
map(lambda x: x * 2 + 10, list)  # 14, 46, 28,54, 44, 58,
26,34, 64

map(lambda w: len(w), 'A lot of cats and dogs
here'.split())  # 1,3,2,4,3,4,4

map(pow, [2, 3, 4], [10, 5, 1, 3])  # 1024,243,4
```

Equality (==) and is

- is operator checks whether 2 arguments refer to the same object
- == operator checks whether 2 arguments have the same value

Equality (==) and is - cont’d

- * Numbers:

```
v1 = 10
v2 = 10
v1 == v2      # compare values, return True
v1 is v2      # probably reference to the same object, so True
print(id(v1), id(v2))   #30779644, 30779644

v3 = 10
v4=11
v3 +=1
v1 == v2      # compare values, return True
v1 is v2      # probably reference to the same object, so True
print(id(v3), id(v4))   #30779632, 3077963
```

Equality (==) and is - cont’d

- This kind of memory sharing we can in the following types as well:

— string
— bool
— None

```
vi ='aaa'; v2 = 'aaa’
v1 == v2              # compare values, return True
v1 is v2              # probably reference to the same object,so True
x = True
y = True
x is y                # probably reference to the same object, so True
```

is and id keywords — cont'd

- unlike the immutable, python will never referee two different mutable objects to the same memory

For example:

```
v1 = [1,2,3]; v2 = [1,2,3]
v1 == v2      # compare values, return True
v1 is v2      # reference to different object, so False
v3=v1
v1 == v3      # compare values, return True
v1 is v3      # reference to the same object, so True
```

Decorators

- Decorator is a feature that extends the functionality of functions without modify it

- Built-in python decorators are @staticmethod and @classmethod

- **@classmethod**
  — The classmethod decorator decorates methods as static methods, like staticmethod. Not like staticmethod, the classmethod receives the class object as the first parameter.

- **@staticmethod**
  - — The staticmethod decorator modifies a method so that it.does not use the self variable. Static method do not have access to any attribute of a specific instance of the class.

Decorators

Demo

Built in decorators - Example

```
class Date(object):
    def __init__(self, day=1, month=1, year=1970):
        self.day = day
        self.month = month
        self.year = year

    @classmethod
    def from_string(cls, date_as_string):
        day, month, year = map(lambda val: int(val), date_as_string.split('-'))
        date1 = cls(day, month, year)
        return date1

    @staticmethod
    def is_date_valid(date_as_string):
        day, month, year = map(lambda val: int(val), date_as_string.split('-'))
        return day <= 31 and month <= 12 and year <= 3999
```

Composition of Decorators

- Define functions inside other functions

```
def print_hello(name):
    def message():
        return "Hello "

    result = message() + name
    return result

print(print_hello ("David"))
```

Outputs: Hello David

Composition of Decorators

Demo

Composition of Decorators — cont’d

- Functions can be passed as parameters to other functions
- Functions can return other functions

```
def message(name):
  return "Hello" + name

def print_message(func):
  print(func("David"))

def bye_message():
  def goodbye_message(name):
    return "Goodbye" + name
  return goodbye_message

print(print_message (message))     # Outputs: Hello David
bye_func = bye_message()
bye_func("David")       # Outputs: Goodbye David
```

Composition of Decorators — cont’d

- Function decorators are simply wrappers to existing functions.
- In this example let's consider a function that substitute space sequences by single space in output string of another function.

```
import re

def get_text(name):
   return "long   sentence      for {0}    ".format(name)
def squeeze_space (func):
   def squeezer(name):
      str=func(name)
      return re.sub(r' +', ' ', str)
   return squeezer

get_text = squeeze_space(get_text)
print(get_text ("John"))   # prints "long sentence for John"


def squeeze_space (func):
   def squeezer(name):
      str=func(name)
      return re.sub(r' +', ' ', str)
   return squeezer

@squeeze_space
def get_text(name):
   return "long sentence for {0} ".format(name)
print(get_text("John")) # prints "long sentence for John"
```

Passing arguments to decorators

- Suppose we want different functions to squeeze custom character

```
import re
def squeeze_char(ch):
    def squeeze_repeat(func):
        def squeezer(name):
            str = func(name)
            return re.sub("{}+".format(ch), ch, str)

        return squeezer

    return squeeze_repeat


@squeeze_char("")
def get_text(name):
    return "long   sentence      for {0}       ".format(name)


@squeeze_char("\n")
def parse_text(text):
    # ……

    print(get_text("John"))  # prints "long sentence for John"
```

Special method names

- A class can implement certain operations that are invoked by special syntax (such as arithmetic operations or iterating) by defining methods with special names.

- Using special methods, your classes can act like sets, like dictionaries, like functions, like iterators, or even like numbers

Overload operators -__str__

- __str__ returns the string representation of an object
- __str__ function can be placed in classes and it is used implicitly in string context like str() or print

```
class Fraction:
   def __init__(self, numerator, denominator):
      self.numerator = numerator
      self.denominator = denominator

   def __str__(self):
      return "{0}/{}".format(self.numerator, self.denominator)

f = Fraction(1,2)
print (f)     # print 1/2
```

Overload operators - Comparing objects

- object.__lt__(self, other) called for x<y
- object.__le__ (self, other) called for x<=y
- object.__eq__(self, other) called for x==y
- object. __ne__(self, other) called for x!=y
- object. __gt__(self, other) called for x>y
- object. __ge__(self, other) called for x>=y

```
class Fraction:
    # ....

    def __lt__(self, other):
        return (self.numerator * other.denominator) < (other.numerator *
self.denominator)

    def __ge__(self, other):
        return not (self < other)
```

Overload operators - Comparing objects — cont’d

```
f1= Fraction(1,2)
f2 = Fraction(1,3)

print(f2 < f1)          # print True
print(f1 >= f2)         # print True
```

Overload operators — more examples

- There are much more operator overloading functions like:

  __iter__ for iterating through — will be discussed later

  __hash__ called on items insertion to dictionary

  __len__ - to support the /en built-in function

  __contains___ - to support in operator

  __nonzero__ called to implement truth value while testing the built-in operation bool() or boolean context

  __add__, __sub__, __ mul__, __ floordiv__, __mod__ for arithmetic operations

  __enter__ and__exit__ - to implement context manager

  And much more...

Iterators

- Iterator is an objects that can be used to go through all its items (with a for loop for example)
- For example:

```
for i in [1,2,3,4]:   or   for i in "python":    or    for i in open("file.txt"):
    print(i)
```

- There are many functions which consume these iterables:
  - for loop
  - sum, min, max built-in functions
  - filter and map built-in functions
  - Comprehension
  - Type conversions to python sequence
  - etc

The Iterator Protocol

- In order to become an iterable object our class should implement the following protocol:

  — It should implement **__iter__** function that returns an iterator

  — The iterator should implement **__next__** function that gives us the next element each time we call it.

  - **__next__** method of python 3, implemented as next method in, python 2
- The built-in function iter takes an iterable object and returns an iterator. Most iter function invocations are implicit, by i Z iterator consumers

The Iterator Protocol — cont’d

```
it = iter([1, 2, 3])
print(it)        # prints <listiterator object at Ox1004ca850>
print( it._next_())  # prints 1
print( it. _next_())     # prints 2
print( it._next_())  # prints 3
print( it._next_())
```

Traceback (most recent call last):

File "<stdin>", line 5, in <module>

StopIteration

The Iterator Protocol V1

Demo

Implement Iterator — Version 1

```
class Yrange:
    def __init__(self, n):
        self.i = 0
        self.in =n

    def __iter__ (self):
        return self

    def __next__(self):
        if self.i < self.n:
            self.i += 1
            return self.i-1
        else:
            raise StopIteration()
```

Implement Iterator — Version 2

```
class Zrange_iter:
    def __init__(self, n):
        self.i = 0
        self.n = n

    def __next__(self):
        if self.i < self.n:
            self.i += 1
            return self.i - 1
        else:
            raise StopIteration()


class Zrange:
    def __init__(self, n):
        self.n = n

    def __iter__(self):
        return Zrange_iter(self.n)
```

Iterators consumers

```
y = Yrange(5)
print(list(y))  # prints [0, 1, 2, 3, 4]
print(list(y))  # prints []

z = Zrange(5)
print(list(z))  # prints [0, 1, 2, 3, 4]
print(list(z))  # prints [0, 1, 2, 3, 4]
print(sum(z))  # prints 10
l1 = [x * 10 for x in z]  # l1 = [0, 10, 20, 30, 40]
l2 = filter(lambda n: n % 2, z)  # l2 = [1, 3]
```

Itertools

- The **itertools** module in Python provides functions for working with iterators efficiently. It offers a range of tools that can be used individually or combined to perform complex operations on iterators. These tools are designed to be fast and memory-efficient.

- For example, suppose you have two lists and you want to multiply their elements. There are different ways to achieve this. One approach is to iterate through the elements of both lists simultaneously and multiply them. Another approach is to use the map function with the mul operator and the two lists as arguments. Let's compare the execution time of each approach.

Itertools – cont’d

Different types of iterators provided by this module are:

- Infinite iterators
- Combinatoric iterators
- Terminating iterators

Itertools – Infinite iterators

Iterator in Python is any Python type that can be used with a ‘for in loop’. Python lists, tuples, dictionaries, and sets are all examples of inbuilt iterators. But it is not necessary that an iterator object has to exhaust, sometimes it can be infinite. Such types of iterators are known as Infinite iterators.

Python provides three types of infinite iterators:

count(start, step): This iterator starts printing from the “start” number and prints infinitely. If steps are mentioned, the numbers are skipped else step is 1 by default. See the below example for its use with for in loop.

Infinite iterators examples

Demo

Infinite iterators examples

cycle(iterable): This iterator prints all values in order from the passed container. It restarts printing from the beginning again when all elements are printed in a cyclic manner.

```
# Python program to demonstrate
# infinite iterators

import itertools

count = 0

# for in loop
for i in itertools.cycle('AB'):
    if count > 7:
        break
    else:
        print(i, end=" ")
        count += 1
```

Infinite iterators examples – cont’d

repeat(val, num): This iterator repeatedly prints the passed value an infinite number of times. If the optional keyword num is mentioned, then it repeatedly prints num number of times.

```
# Python code to demonstrate the working of
# repeat()

# importing "itertools" for iterator operations
import itertools

# using repeat() to repeatedly print number
print("Printing the numbers repeatedly : ")
print(list(itertools.repeat(25, 4)))
```

Itertools – Combinatoric iterators

The recursive generators that are used to simplify combinatorial constructs such as permutations, combinations, and Cartesian products are called combinatoric iterators.

In Python there are 4 combinatoric iterators:

Product(): This tool computes the cartesian product of input iterables. To compute the product of an iterable with itself, we use the optional repeat keyword argument to specify the number of repetitions. The output of this function is tuples in sorted order.

Combinatoric iterators examples

Demo

Combinatoric combinatoric iterators

Permutations(): Permutations() as the name speaks for itself is used to generate all possible permutations of an iterable. All elements are treated as unique based on their position and not their values. This function takes an iterable and group_size, if the value of group_size is not specified or is equal to None then the value of group_size becomes the length of the iterable.

```
# import the product function from itertools module
from itertools import permutations

print("All the permutations of the given list is:")
print(list(permutations([1, 'geeks'], 2)))
print()

print("All the permutations of the given string is:")
print(list(permutations('AB')))
print()

print("All the permutations of the given container is:")
print(list(permutations(range(3), 2)))
```

Combinatoric combinatoric iterators

Combinations(): This iterator prints all the possible combinations(without replacement) of the container passed in arguments in the specified group size in sorted order.

```
# import combinations from itertools module

from itertools import combinations

print("All the combination of list in sorted order(without replacement) is:")
print(list(combinations(['A', 2], 2)))
print()

print("All the combination of string in sorted order(without replacement) is:")
print(list(combinations('AB', 2)))
print()

print("All the combination of list in sorted order(without replacement) is:")
print(list(combinations(range(2), 1)))
```

Combinatoric combinatoric iterators

Combinations_with_replacement(): This function returns a subsequence of length n from the elements of the iterable where n is the argument that the function takes determining the length of the subsequences generated by the function. Individual elements may repeat itself in combinations_with_replacement function.

```
# import combinations from itertools module

from itertools import combinations_with_replacement

print("All the combination of string in sorted order(with replacement) is:")
print(list(combinations_with_replacement("AB", 2)))
print()

print("All the combination of list in sorted order(with replacement) is:")
print(list(combinations_with_replacement([1, 2], 2)))
print()

print("All the combination of container in sorted order(with replacement) is:")
print(list(combinations_with_replacement(range(2), 1)))
```

Itertools – Terminating iterators

Terminating iterators are used to work on the short input sequences and produce the output based on the functionality of the method used.

Different types of terminating iterators are:

accumulate(iter, func): This iterator takes two arguments, iterable target and the function which would be followed at each iteration of value in target. If no function is passed, addition takes place by default. If the input iterable is empty, the output iterable will also be empty.

Terminating iterators examples

Demo

Terminating iterators examples

chain(iter1, iter2..): This function is used to print all the values in iterable targets one after another mentioned in its arguments.

```
# Python code to demonstrate the working of
# and chain()


import itertools

# initializing list 1
li1 = [1, 4, 5, 7]

# initializing list 2
li2 = [1, 6, 5, 9]

# initializing list 3
li3 = [8, 10, 5, 4]

# using chain() to print all elements of lists
print("All values in mentioned chain are : ", end="")
print(list(itertools.chain(li1, li2, li3)))
```

chain.from_iterable(): This function is implemented similarly as a chain() but the argument here is a list of lists or any other iterable container.

```
# Python code to demonstrate the working of
# chain.from_iterable()


import itertools

# initializing list 1
li1 = [1, 4, 5, 7]

# initializing list 2
li2 = [1, 6, 5, 9]

# initializing list 3
li3 = [8, 10, 5, 4]

# initializing list of list
li4 = [li1, li2, li3]

# using chain.from_iterable() to print all elements of lists
print("All values in mentioned chain are : ", end="")
print(list(itertools.chain.from_iterable(li4)))
```

Terminating iterators examples

compress(iter, selector): This iterator selectively picks the values to print from the passed container according to the boolean list value passed as other arguments. The arguments corresponding to boolean true are printed else all are skipped.

```
# Python code to demonstrate the working of
# and compress()


import itertools

# using compress() selectively print data values
print("The compressed values in string are : ", end="")
print(list(itertools.compress('GEEKSFORGEEKS', [
    1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0])))
```

dropwhile(func, seq): This iterator starts printing the characters only after the func. in argument returns false for the first time.

```
# Python code to demonstrate the working of
# dropwhile()


import itertools

# initializing list
li = [2, 4, 5, 7, 8]

# using dropwhile() to start displaying after condition is false
print("The values after condition returns false : ", end="")
print(list(itertools.dropwhile(lambda x: x % 2 == 0, li)))
```

Terminating iterators examples

filterfalse(func, seq): As the name suggests, this iterator prints only values that return false for the passed function.

islice(iterable, start, stop, step): This iterator selectively prints the values mentioned in its iterable container passed as argument. This iterator takes 4 arguments, iterable container, starting pos., ending position and step.

```
# Python code to demonstrate the working of
# filterfalse()


import itertools

# initializing list
li = [2, 4, 5, 7, 8]

# using filterfalse() to print false values
print("The values that return false to function are : ", end="")
print(list(itertools.filterfalse(lambda x: x % 2 == 0, li)))
```

```
# Python code to demonstrate the working of
# islice()


import itertools

# initializing list
li = [2, 4, 5, 7, 8, 10, 20]

# using islice() to slice the list acc. to need
# starts printing from 2nd index till 6th skipping 2
print("The sliced list values are : ", end="")
print(list(itertools.islice(li, 1, 6, 2)))
```

Terminating iterators examples

starmap(func., tuple list): This iterator takes a function and tuple list as argument and returns the value according to the function from each tuple of the list.

```
# Python code to demonstrate the working of
# starmap()


import itertools

# initializing tuple list
li = [(1, 10, 5), (8, 4, 1), (5, 4, 9), (11, 10, 1)]

# using starmap() for selection value acc. to function
# selects min of all tuple values
print("The values acc. to function are : ", end="")
print(list(itertools.starmap(min, li)))
```

takewhile(func, iterable): This iterator is the opposite of dropwhile(), it prints the values till the function returns false for 1st time.

```
# Python code to demonstrate the working of
# takewhile()


import itertools

# initializing list
li = [2, 4, 6, 7, 8, 10, 20]

# using takewhile() to print values till condition is false.
print("The list values till 1st false value are : ", end="")
print(list(itertools.takewhile(lambda x: x % 2 == 0, li)))
```

Terminating iterators examples

tee(iterator, count):- This iterator splits the container into a number of iterators mentioned in the argument.

```
# Python code to demonstrate the working of
# tee()


import itertools

# initializing list
li = [2, 4, 6, 7, 8, 10, 20]

# storing list in iterator
iti = iter(li)

# using tee() to make a list of iterators
# makes list of 3 iterators having same values.
it = itertools.tee(iti, 3)

# printing the values of iterators
print("The iterators are : ")
for i in range(0, 3):
    print(list(it[i]))
```

zip_longest(iterable1, iterable2, fillval): This iterator prints the values of iterables alternatively in sequence. If one of the iterables is printed fully, the remaining values are filled by the values assigned to fillvalue.

```
# Python code to demonstrate the working of
# zip_longest()


import itertools

# using zip_longest() to combine two iterables.
print("The combined values of iterables is  : ")
print(*(itertools.zip_longest('GesoGes', 'ekfrek',
fillvalue='_')))
```

Generators

- Generators are functions allow as to declare a function that behaves like an iterator, i.e. it can be used in a for loop

- Generators simplifies creation of iterators. A generator is a function that produces a sequence of results instead of a single value.

- Generators are functions that have a **yield** statement

Implement generator

Demo

For example:

```
def gen_range(n):
    i = 0
    while i < n:
        yield i
        i +=1

res_iter = gen_range(5)
for n in res_iter:
    print(n)
```

Generators consumers

```
def gen(n):
    for i in range(n+1):
        yield i

print(sum(gen(10)))       #55

for el in gen(4):         #01234
    print (el)

first7Values = gen(7)
lst = [x*x for x in first7Values]
print (lst)               #[0149 16 25 36 49]
```

Python Garbage Collection

- Python's memory allocation and deallocation method is automatic.
- Python uses two strategies for memory allocation™ reference counting and garbage collection.
- Prior to Python version 2.0, the Python interpreter only used reference counting for memory management.

Reference counting

- Reference counting works by counting the number of times an object is referenced by other objects in the system.
- When references to an object are removed, the reference count for an object is decremented.
- When the reference count becomes zero the object is deallocated, including invoking __del__ method on objects
- Reference counting is extremely efficient but it does not always works
  - One of the biggest problems in reference counting is reference cycles.

Reference counting — reference cycles

- A reference cycles prevents from objects' reference count to drops down to zero, yet there is no way to reach the objects.

- For example:

```
def make_cycle():
    l=[]
    l.append(l)
    make_cycle()
```

AGC - Automatic Garbage Collection

- The main role of AGC in python is to detect reference cycles for python sequences and user objects and to deallocate them
- The automatic garbage collection has been included in Python since version 2.0.
- A reference-counting scheme collects objects as soon.as they become unreachable
- The time of collection of objects with circular references is unknown.
- Do not depend on immediate finalization of objects when” - they become unreachable

Python gc - Manual Garbage Collection

- For some programs, especially long running server applications or embedded applications, automatic garbage collection may not be sufficient

- The garbage collection can be invoked manually with **gc.collect()** function, which returns the number of objects it has collected and deallocated

GC example

Demo

gc example

```
import gc

def make_cycle():
    l = []
    l.append(l)

def main():
    collected = gc.collect()
    print("Garbage collector: collected {} objects.".format(collected))
    print("Creating cycles...")
    for i in range(10):
        make_cycle()
    collected = gc.collect()
    print("Garbage collector: collected {} objects.".format(collected))

main()
```

Random

- Python's random module provides functions to generate random values. It is commonly used in various applications such as games, simulations, and statistical analysis. Let's explore some commonly used functions:

- random.random(): This function returns a random float number between 0 and 1. It is often used when you need a random decimal number for a particular purpose.

```
import random

random_number = random.random()
print(random_number)
```

Random – cont’d

- random.randint(a, b): This function returns a random integer between a and b, inclusive. It is useful when you need a random whole number within a specific range.

```
import random

random_integer = random.randint(1, 10)
print(random_integer)
```

- random.choice(sequence): This function returns a random element from the given sequence, such as a list, tuple, or string. It allows you to select a random item from a collection.

```
import random

fruits = ['apple', 'banana', 'orange', 'kiwi']
random_fruit = random.choice(fruits)
print(random_fruit)
```

Random – cont’d

- **random.shuffle(sequence):** This function shuffles the elements in the given sequence in-place. It is useful when you want to randomize the order of a list or any other sequence.

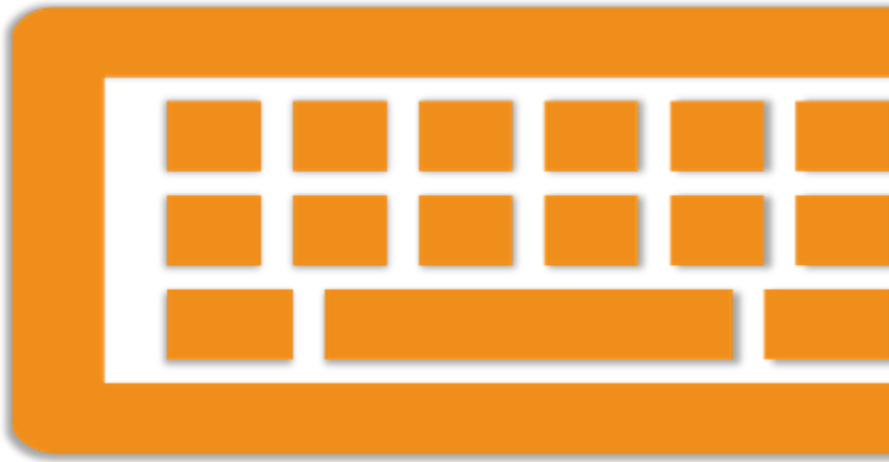
```
import random

cards = ['Ace', '2', '3', '4', '5', '6', '7', '8', '9', '10', 'Jack', 'Queen', 'King']
random.shuffle(cards)
print(cards)
```

- These functions are just a glimpse of what the random module offers. There are additional functions available, such as **random.uniform(), random.sample(),** and **random.randrange(),** which provide further flexibility in generating random values. By utilizing these functions effectively, students can add randomness to their programs, making them more dynamic and realistic.

Lab 01

Lab



Questions