1 Series 01: variables, expressions and statements

1.1 ISBN

```
# read first nine digits of an ISBN-10 code
x1 = int(input())
x2 = int(input())
x3 = int(input())
x4 = int(input())
x5 = int(input())
x6 = int(input())
x7 = int(input())
x8 = int(input())
x9 = int(input())

# compute check digit
x10 = (
    x1 + 2 * x2 + 3 * x3 + 4 * x4 + 5 * x5 + 6 * x6 + 7 * x7 + 8 * x8 + 9 * x9
) % 11

# print check digit
print(x10)
```

1.2 Sum of two integers

```
# read two terms and convert them to integers
term1 = int(input('Give an integer: '))
term2 = int(input('Give another integer: '))

# compute sum of two integers
# NOTE: we do not use the name "sum" for the variable "sum" because this is the
#       name of a built-in function in Python
total = term1 + term2

# write sum to output
print(total)
```

1.3 Heartbeats

```
# read creature features
creatures = input()                # name of a creature (plural)
heartRate = int(input())           # heart rate (per minute)
longevity = int(input())           # longevity (in years)

# determine number of heartbeats in a lifetime
minutesPerYear = 60 * 24 * 365
heartbeats = heartRate * minutesPerYear * longevity

# output number of heartbeats in a lifetime
template = '{} have {:.2f} billion heartbeats'
print(template.format(creatures, heartbeats / 1e9))
```

1.4 The diatomist

```
import math

# read diameter of smaller and larger circles
r = float(input())
R = float(input())
```

```
# estimate number of smaller circles that fit in larger circle
count = math.floor(0.83 * (R ** 2 / r ** 2) - 1.9)

# determine coverage of larger circle
area_large = math.pi * R ** 2
area_small = math.pi * r ** 2
coverage = (count * area_small) / area_large * 100

# output number of circles and coverage of larger circle
template = '{} smaller circles cover {:.2f}% of the larger circle'
print(template.format(count, coverage))
```

1.5 Timekeeping on Mars

```
# read number of sol
sol = int(input())

# express number of sol in seconds
seconds = int(sol * (((24 * 60) + 39) * 60 + 35.244))

# convert seconds into minutes and seconds
minutes = seconds // 60 floor
seconds %= 60 通过%= 更新得到想要的结果

# convert minutes into hours and minutes
hours = minutes // 60
minutes %= 60

# convert hours into days and hours
days = hours // 24
hours %= 24

# output conversion of sol into days, hours, minutes and seconds
template = '{} sols = {} days, {} hours, {} minutes and {} seconds'
print(template.format(sol, days, hours, minutes, seconds))
```

1.6 Clock hands

```
# read time from input
hours = int(input())
minutes = int(input())

"""
# determine angle that minute hand makes (from 12 o'clock)
angle_minute = minutes / 60

# determine angle that hour hand makes (from 12 o'clock); take into account that
# the hour hand also progresses as the minutes pass by
angle_hour = (hours % 12 + angle_minute) / 12

# determine one of the angles between both hands
angle_hands = (360 * (angle_hour - angle_minute)) % 360
"""

# some simple arithmetic reduces the above three statements to
angle_hands = (30 * hours - 5.5 * minutes) % 360

# determine smallest angle between both hands
angle_hands = min(angle_hands, 360 - angle_hands)

# output smallest between both hands
template = 'At {:02d}:{:02d} both hands form an angle of {:.1f}°.'
print(template.format(hours, minutes, angle_hands))
```

2 Series 02: conditional statements

2.1 ISBN

```
# read ten digits of an ISBN-10 code (each on a separate line)
x1 = int(input())
x2 = int(input())
x3 = int(input())
x4 = int(input())
x5 = int(input())
x6 = int(input())
x7 = int(input())
x8 = int(input())
x9 = int(input())
x10 = int(input())

# compute check digit
checkdigit = (
    x1 + 2 * x2 + 3 * x3 + 4 * x4 + 5 * x5 + 6 * x6 + 7 * x7 + 8 * x8 + 9 * x9
) % 11

# check correctness of check digit
print('OK' if x10 == checkdigit else 'WRONG')

"""
alternative solution:

if x10 == checkdigit:
    print('OK')
else:
    print('WRONG')
"""
```

2.2 Personal warmth

```
import math

# read body temperature
body_temperature = float(input())

# make estimate of e
estimate = 100 / body_temperature

# make diagnosis concerning body temperature
eps = 0.1
if estimate < math.e - eps:
    diagnosis = 'you have a fever'
elif estimate > math.e + eps:
    diagnosis = 'you have hypothermia'
else:
    diagnosis = 'you have a normal body temperature'

# output diagnosis
print(diagnosis)
```

2.3 Birthstones

```
# read given birthstone
birthstone = input()

# define mapping between birthstones and months of birth
months = {
```

```
    'amethyst': 'february',
    'alexandrite': 'june',
    'aquamarine': 'march',
    'bloodstone': 'march',
    'carnelian': 'july',
    'chrysoprase': 'may',
    'citrine': 'november',
    'diamond': 'april',
    'emerald': 'may',
    'garnet': 'january',
    'lapis lazuli': ['september', 'december'],
    'moonstone': 'june',
    'pearl': 'june',
    'peridot': 'august',
    'opal': 'october',
    'rock crystal': 'april',
    'ruby': 'july',
    'sapphire': 'september',
    'sardonyx': 'august',
    'spinel': 'august',
    'tanzanite': 'december',
    'topaz': 'november',
    'tourmaline': 'october',
    'turquoise': 'december',
    'zircon': 'december',
}

# determine month(s) of birth for given birthstone
month = months[birthstone]

# print mapping between given birthstone and month(s) of birth
print('{} is a birthstone of the month {}'.format(
    birthstone,
    month if isinstance(month, str) else ' or '.join(month)
))
```

2.4 Counterfeiting

```
# find group that contains the counterfeit coin based on the first weighing:
# group 0 = 1-2-3; group 1 = 4-5-6; group 2 = 7-8-9
weighing = input()
group = 0 if weighing == 'right' else (1 if weighing == 'left' else 2)

# determine which coin in the group is counterfeit
weighing = input()
coin = 1 if weighing == 'right' else (2 if weighing == 'left' else 3)

# indicate which coin is counterfeit
print('coin #{} is counterfeit'.format(3 * group + coin))
```

2.5 The two towers

```
# read outcome of throw of both coins; outcomes are converted to Boolean values
# (head -> True, tail -> False) in order to ease the implementation
coin1 = input() == 'head'
coin2 = input() == 'head'

# read which scientist will say the same as his own outcome
same = input()

# determine response of both scientists based on the outcome of their own throws
# and the agreement who will say the same and who will say the opposite
if same == 'first':
    coin1, coin2 = coin1, not coin2
```

```
else:
    coin1, coin2 = not coin1, coin2

# output response of first scientist
print('head' if coin1 else 'tail')

# output response of second scientist
print('head' if coin2 else 'tail')
```

2.6 Knight move

```
# read two given position on chess board
position1 = input()
position2 = input()

# decompose positions into row and column indices
col1, row1 = position1
col2, row2 = position2

# convert row indices into integers
row1, row2 = int(row1), int(row2)

# convert column indices into integers (zero-based)
col1 = ord(col1) - ord('a')
col2 = ord(col2) - ord('a')

# determine whether or not knight can jump between two given positions: this is
# the case if it jumps over one column and two rows or one row and two columns
print('a knight can{} jump from {} to {}'.format(
    '' if {abs(row1 - row2), abs(col1 - col2)} == {1, 2} else 'not',
    position1,
    position2
))
```

3 Series 03: loops

3.1 ISBN

```
# read the first digit of a ISBN-10 code
# NOTE: at this point we cannot assume that the first line of the first ISBN-10
#       code contains a digit, since it may also contain the word stop
x1 = input()

while x1 != 'stop':

    # read the next eight digits and compute check digit
    checkdigit = int(x1)
    for i in range(2, 10):
        checkdigit += i * int(input())
    checkdigit %= 11

    # read check digit and test its correctness
    print('OK' if checkdigit == int(input()) else 'WRONG')

    # read first digit of next ISBN-10 code
    # NOTE: at this point we cannot assume that the first line of the next
    #       ISBN-10 code contains a digit, since it may also contain the word
    #       stop
    x1 = input()
```

3.2 Payslip

```
# read random number
random = int(input())

# intialize total salary with random number
total = random

# one by one process salary of each worker
salary, workers = input(), 0
while salary != 'stop':

    # add salary of worker to total salary
    workers += 1
    total += int(salary)

    # output total salary as whispered by worker
    print('worker #{} whispers {}'.format(workers, total))

    # read salary of next worker
    salary = input()

# output average salary
print('average salary: {:.2f}'.format((total - random) / workers))
```

3.3 Conan the Bacterium

```
# read parameters of experiments
a = int(input())
b = int(input())
n = int(input())
t = int(input())

# determine number of bacteria z obtained after growing a single bacterial
# strain for n seconds
z = 1
```

```
for _ in range(n):
    z = a * z + b

# output number of cells obtained after first experiment
print('experiment #1: {} cells after {} seconds'.format(z, n))

# determine minimal number of seconds needed to grow at least z bacteria
# starting from t bacteria
seconds = 0
while t < z:
    seconds += 1
    t = a * t + b

# output how long cells have to grow during second experiment
print('experiment #2: {} cells after {} seconds'.format(t, seconds))
```

3.4 Heat wave

```
# initialize variables with properties about sequence of successive warm days
summer_days = 0      # number of successive days with temperature above 25 °C
tropical_days = 0    # number of days in sequence with temperature above 30 °C

# no heat wave observed until a sequence of successive days is found that meets
# the criteria of a heat wave
heat_wave = False

# iterate days and determine the length of the current sequence of successive
# days with a temperature above 25 °C, and the number of days in that sequence
# with a temperature above 30 °C
line = input()
while not heat_wave and line != 'stop':

    # line contains a temperature
    temperature = float(line)

    if temperature >= 25:                    # extend sequence above 25 °C

        summer_days += 1
        if temperature >= 30:
            tropical_days += 1

        # determine if conditions of heat wave have been met
        if summer_days >= 5 and tropical_days >= 3:
            heat_wave = True

    else:                                    # start new sequence above 25 °C

        summer_days = 0
        tropical_days = 0

    # read next line from input
    line = input()

# output whether or not a heat wave was observed during the given period
print('heat wave' if heat_wave else 'no heat wave')
```

3.5 Challenger or crack

```
# read number of questions in the round
questions = int(input())

# process all answers in the question round
score_challenger, score_crack = 0, 0
for _ in range(questions):
```

```
    # process next question: read correct answer and given answers
    correct_answer = input()
    answer_challenger = input()
    answer_crack = input()

    # determine if challenger scores a point on the question
    if answer_challenger == correct_answer:
        score_challenger += 1

    # determine if crack scores a point on the question
    if (answer_crack == 'correct') == (answer_challenger == correct_answer):
        score_crack += 1

# define a very small value that is used to counter rounding errors when working
# with floating point numbers
eps = 1e-6

# determine outcome of the round
if score_crack < (questions / 2) - eps or score_challenger > score_crack:
    template = 'challenger wins {challenger} points against {crack}'
elif score_crack == score_challenger:
    template = 'ex aequo: both contestants score {crack} points'
else:
    template = 'crack wins {crack} points against {challenger}'

# uitkomst van de ronde uitschrijven
print(template.format(crack=score_crack, challenger=score_challenger))
```

3.6 Three wise men

```
import math

# read total price for gifts
price = float(input())

# multiply price by 100 to make computations in integer space
integer_price = round(price * 100)

# define output template
template = '${0:.2f} + ${1:.2f} + ${2:.2f} = ${0:.2f} x ${1:.2f} x ${2:.2f} = ${3:.2f}'

# flag to stop searching as soon as solution has been found
found = False

# determine individual prices that meet criteria of addition and multiplication
sum1 = integer_price
prod1 = sum1 * 10000
a = 1
while not found and a < min(sum1 // 3, math.floor(prod1 ** (1 / 3))) + 1:
    if not prod1 % a:
        sum2 = sum1 - a
        prod2 = prod1 // a
        b = a
        while not found and b < min(sum2 // 2, math.floor(prod2 ** 0.5)) + 1:
            if not prod2 % b:
                c = prod2 // b
                if b + c == sum2:
                    found = True
                    print(template.format(a / 100, b / 100, c / 100, price))
            b += 1
    a += 1
```

4 Series 04: strings

4.1 ISBN

```
# read first ISBN-10 code (or the word stop)
code = input()

# read successive ISBN-10 codes until line containing "stop" is read
while code != 'stop':

    # compute check digit
    check_digit = sum((i + 1) * int(code[i]) for i in range(9)) % 11

    """
    # compute check digit: alternatieve oplossing

    check_digit = int(code[0])
    for i in range(2, 10):
        check_digit += i * int(code[i - 1])
    check_digit %= 11
    """

    # extract check digit from ISBN-10 code
    x10 = code[9]

    # check whether computed and extracted check digits are the same
    if (check_digit == 10 and x10 == 'X') or x10 == str(check_digit):
        print('OK')
    else:
        print('WRONG')

    # read next ISBN-10 code (or the word stop)
    code = input()
```

4.2 All the king's wine

```
# initialize buttle number
bottle = 0

# increment bottle number according to value assigned to each prisoner
for _ in range(int(input())):
    bottle += 2 ** (ord(input()) - ord('A'))

# return which bottle has been poisoned
print('Bottle #{} is poisoned.'.format(bottle))
```

4.3 Number walks

```
from math import sin, cos, radians

# read number
number = input()

# initialize starting position
x, y = 0.0, 0.0

# take steps in directions indicated by digits in the number
for digit in number:
    if digit.isdigit():
        angle = radians(int(digit) * 36)
        x += sin(angle)
        y += cos(angle)
```

```
# ouput end position of number walk
template = 'Number {} walks to position ({:.2f}, {:.2f}).'
print(template.format(number, x, y))
```

4.4 Reading a pitch

```
# read the slogan (string)
slogan = input()

# read the starting position and the step size (integers)
position = int(input())
step = int(input())

# find and output the hidden message
print(''.join(
    slogan[(position + i * step) % len(slogan)] for i in range(len(slogan))
))

"""
# alternative solution:

# find the hidden message
message = ''
for i in range(len(slogan)):
    message += slogan[(position + i * step) % len(slogan)]

# output the hidden message
print(message)
"""
```

4.5 Wow! signal

```
# determine number of lines to be scanned
lines = int(input())

# scan lines one by one
for _ in range(lines):

    # read next line
    line = input()

    # scan line by determining groups that represent a wow signal
    scan, group = '', ''
    lowdigits, lowercase, uppercase = False, False, False
    for character in line:

        if character.isalnum():

            # extend group
            group += character

            # determine kind of character
            if character.islower():
                lowercase = True
            elif character.isupper():
                uppercase = True
            elif character.isdigit() and int(character) < 5:
                lowdigits = True

        else:

            # determine whether or not group represents wow signal
            if not lowdigits and (not uppercase or not lowercase):
```

```
                scan += group
            else:
                scan += '.' * len(group)

            # start observing a new group
            group = ''
            lowdigits, lowercase, uppercase = False, False, False

            # put a dot at the current position
            scan += '.'

    # determine whether or not group represents wow signal
    if not lowdigits and (not uppercase or not lowercase):
        scan += group
    else:
        scan += '.' * len(group)

    # output scanned line
    print(scan)
```

4.6 The missing number 值得练习

截短型问题

```
# read sequence of digits (as a string)
sequence = input()

# initially no missing number has been found
missing = None

# initially we assume that first number has a single digit
digits = 1                                  If the length of comparing string is longer than the remaining,
                                            then no need continue to do loop
while missing is None and digits <= len(sequence) // 2:

    # first number is composed of first "digits" digits from the sequence
    next = int(sequence[:digits])

    # remaining sequence follows the first number
    remaining = sequence[digits:]

    # find missing number in sequence based on the chosen first number
    while remaining:

        # determine next number that follows previous number in the sequence
        next += 1

        if remaining.startswith(str(next)):
            # remove the next number from the start of the remaining sequence
            remaining = remaining[len(str(next)):]   Remaining 的改变, remaining of remaining, 循环作用. 注意不要
        elif missing is None:                        使用seq[], 这样无法起到循环的作用.
            # remember first missing number that was found
            missing = next   起到两个作用:
        else:                1. 记录missing number 2. 标志着至少找到一个
            # second missing number found, so this is not a valid sequence for
            # the current number of digits; the missing number that was already
            # found turns out to be invalid
            remaining = ''
            missing = None   清空remaining, 增加digit个数, 重新寻找

    # process sequence with an extra digit for the first number
    digits += 1

# output missing number
print(missing if missing is not None else 'no missing number')
```

5 Series 05: functions

5.1 ISBN

```
def isISBN(code):

    """
    Checks whether or not the given ISBN-10 code is valid.

    >>> isISBN('9971502100')
    True
    >>> isISBN('9971502108')
    False
    """

    # note: isinstance is a Python built-in function that returns a Boolean
    #       value that indicates whether or not the first argument is an object
    #       that has a data type equal to the second argument passed to the
    #       function
    if not (
        isinstance(code, str) and  # code must be a string
        len(code) == 10 and        # code must contains 10 characters
        code[:9].isdigit()         # first nine characters must be digits
    ):
        return False

    # check the check digit
    return checkdigit(code) == code[-1]

def checkdigit(code):

    """
    >>> checkdigit('997150210')
    '0'
    >>> checkdigit('938389293')
    '5'
    """

    # compute check digit
    check = sum((i + 1) * int(code[i]) for i in range(9)) % 11

    # convert check digit into string representation
    return 'X' if check == 10 else str(check)

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

5.2 Noah's headache

```
def split(species):

    """
    Splits the given string in a prefix and a suffix, x, where the prefix is
    formed by the longest sequence of consonants at the start of the word.

    >>> split('scheep')
    ('sch', 'eep')
    >>> split('goat')
    ('g', 'oat')
    """

    # find position of first vowel
    pos = 0
    while pos < len(species) and species[pos].lower() not in 'aeiou':
```

```
        pos += 1

    # split species name in prefix and suffix
    return species[:pos], species[pos:]

def hybridize(species1, species2):

    """
    Returns a tuple containing two strings. The first element of the tuple is
    formed by concatenating the prefix of the first given string and the suffix
    of the second given string. The second element of the tuple is formed by
    concatenating the prefix of the second given string and the suffix of the
    first given string.

    >>> hybridize('goat', 'sheep')
    ('geep', 'shoat')
    >>> hybridize('lion', 'tiger')
    ('jeopard', 'laguar')
    >>> hybridize('schnauzer', 'poodle')
    ('schnoodle', 'pauzer')
    """

    # split species names in prefix and suffix
    prefix1, suffix1 = split(species1)
    prefix2, suffix2 = split(species2)

    # hybridize the species names
    return prefix1 + suffix2, prefix2 + suffix1

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

5.3 Looking up

```
def swap(cards):

    """
    >>> swap('FBFFFBFFBF')
    'BFBBBFBBFB'
    >>> swap('BFFBFBFFFBFBBBFBBBBFF')
    'FBBFBFBBBFBFFFBFFFFBB'
    >>> swap('FFBFBFBFBFBFBFBBFBFBFBFBBFBFBBFBF')
    'BBFBFBFBFBFBFBFFBFBFBFBFFBFBFFBFB'
    """

    # swap all cards
    return ''.join('B' if card == 'F' else 'F' for card in cards)

def next(cards):

    """
    >>> next('FBFFFBFFBF')
    'FFBBBFBBFF'
    >>> next('BFFBFBFFFBFBBBFBBBBFF')
    'FBBFBFBBBFBFFFBFFFFFF'
    >>> next('FFBFBFBFBFBFBFBBFBFBFBFBBFBFBBFBF')
    'FFFBFBFBFBFBFBFFBFBFBFBFFBFBFFBFF'
    """

    # find first card that is face down
    first = cards.find('B')

    # we only need to swap cards if there are cards that face down
    if first >= 0:

        # find last card that is face down
```

```
        last = cards.rfind('B') + 1

        # swap cards between first and last (including outer cards)
        turned = swap(cards[first:last])

        # determine new configuration of the cards
        cards = cards[:first] + turned + cards[last:]

    # return new configuration of cards after swapping
    return cards

def turns(cards):

    """
    >>> turns('FBFFFBFFBF')
    3
    >>> turns('BFFBFBFFFBFBBBFBBBBFF')
    6
    >>> turns('FFBFBFBFBFBFBFBBFBFBFBFBBFBFBBFBF')
    14
    """

    # keep on turning cards as long as there are still cards that are face down
    # and count the number of turns needed to bring all cards face up
    turns = 0
    while 'B' in cards:
        turns += 1
        cards = next(cards)

    # return the number of turns needed to bring all cards face up
    return turns

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

5.4 Ergonomics

```
def position(letter):

    """
    >>> row, col = position('K')
    >>> row
    0
    >>> col
    10
    >>> row, col = position('q')
    >>> row
    1
    >>> col
    3
    """

    # determine position of letter in alfabet
    pos = ord(letter.upper()) - ord('A')

    # return row and column index
    return pos // 13, pos % 13

def shift(letter1, letter2):

    """
    >>> shift('K', 'q')
    8
    >>> shift('f', 'e')
    1
    """
```

```
    # determine position of letters on ouija board (row and colom index)
    row1, col1 = position(letter1)
    row2, col2 = position(letter2)

    # compute distance between two letters
    return abs(row1 - row2) + abs(col1 - col2)

def ergonomics(word):

    """
    >>> ergonomics('FEED')
    2
    >>> ergonomics('MAMA')
    36
    >>> ergonomics('feeders')
    5
    >>> ergonomics('layaway')
    67
    >>> ergonomics('disestablismentarianism')
    113
    >>> ergonomics('electroencephalographic')
    108
    """

    # compute total distance travelled on ouija board
    distance = 0
    for i in range(len(word) - 1):
        distance += shift(word[i], word[i + 1])

    # return total distance travelled
    return distance

    """
    # alternative solution using generator

    return sum(shift(word[i], word[i + 1]) for i in range(len(word) - 1))
    """

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

5.5 Phone neighbours

```
def digits(phone):

    """
    >>> digits('0472/91.39.17')
    '0472913917'
    >>> digits('++32 (0)9 264 4779')
    '32092644779'
    """

    # extract digits from the given phone number
    return ''.join(c for c in phone if c.isdigit())

def replace(phone, number):

    """
    >>> replace('0472/91.39.17', 1234567890)
    '1234/56.78.90'
    >>> replace('++32 (0)9 264 4779', 123456789)
    '++00 (1)2 345 6789'
    """

    # convert the given number into a string and add leading zeros until it
```

```
    # has at least the same number of digits as the given phone number
    number = str(number).zfill(len(digits(phone)))

    # create iterator that traverses the digits of the number
    digit = iter(number)

    # replace digits of given phone number by successive digits of the number
    return ''.join(next(digit) if c.isdigit() else c for c in phone)

def neighbour(phone, delta):

    """
    >>> neighbour('0472/91.39.17', 1)
    '0472/91.39.18'
    >>> neighbour('0472/91.39.17', -1)
    '0472/91.39.16'
    """

    # replace digits of the given phone numbers with those of the number formed
    # by the digits in the phone number plus the given delta
    return replace(phone, int(digits(phone)) + delta)

def upstairsNeighbour(phone):

    """
    >>> upstairsNeighbour('0472/91.39.17')
    '0472/91.39.18'
    >>> upstairsNeighbour('++32 (0)9 264 4779')
    '++32 (0)9 264 4780'
    """

    # replace digits of given phone number by those of its upstairs neighbour
    return neighbour(phone, 1)

def downstairsNeighbour(phone):

    """
    >>> downstairsNeighbour('0472/91.39.17')
    '0472/91.39.16'
    >>> downstairsNeighbour('++32 (0)9 264 4779')
    '++32 (0)9 264 4778'
    """

    # replace digits of given phone number by those of its downstairs neighbour
    return neighbour(phone, -1)

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

5.6 Turkey Irish

```
def isVowel(character):

    """
    >>> isVowel('a')
    True
    >>> isVowel('c')
    False
    >>> isVowel('E')
    True
    """

    # determine if character is a vowel
    return character.lower() in 'aeiou'

def encode(sentence):
```

```
    """
    >>> encode('Fabiano')
    'Fababababianabo'
    >>> encode('CIA-agent')
    'CAbiA-abagabent'
    """

    # traverse letters one by one and determine groups of vowels
    group, ciphertext = '', ''
    for character in sentence:

        if isVowel(character):
            # extend group of vowels
            group += character
        else:
            # prefix group of vowels with string "ab" and then start looking for
            # a new group of vowels
            if group:
                prefix = 'Ab' if group[0].isupper() else 'ab'
                ciphertext += prefix + group[0].lower() + group[1:]
                group = ''

            # copy non-vowel to ciphertext
            ciphertext += character

    # prefix last group of vowels (if any) with string "ab"
    if group:
        prefix = 'Ab' if group[0].isupper() else 'ab'
        ciphertext += prefix + group[0].lower() + group[1:]

    # return encoded sentence
    return ciphertext

def decode(sentence):

    """
    >>> decode('Fababababianabo')
    'Fabiano'
    >>> decode('CAbiA-abagabent')
    'CIA-agent'
    """

    # traverse characters in ciphertext one by one
    plaintext, i = '', 0
    while i < len(sentence):

        if sentence[i:i + 2].lower() == 'ab':

            # add first vowel after "ab" to plaintext (with original case)
            vowel = sentence[i + 2]
            if sentence[i:i + 2] == 'Ab':
                vowel = vowel.upper()
            plaintext += vowel

            # add next vowels to plaintext, as they are not part of an added
            # "ab" string by definition
            i += 3
            while i < len(sentence) and isVowel(sentence[i]):
                plaintext += sentence[i]
                i += 1

        else:

            # add character to plaintext
            plaintext += sentence[i]
            i += 1

    # return plaintext
```

```
    return plaintext

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

6 Series 06: lists and tuples

6.1 ISBN

```
def isISBN(code):

    """
    Checks whether or not the given ISBN-10 code is valid.

    >>> isISBN('9-9715-0210-0')
    True
    >>> isISBN('997-150-210-0')
    False
    >>> isISBN('9-9715-0210-8')
    False
    """

    # check if the given code is a string
    if not isinstance(code, str):
        return False

    # checks whether dashes are at the correct positions and whether each group
    # has the correct number of digits
    groups = code.split('-')
    if tuple(len(e) for e in groups) != (1, 4, 4, 1):
        return False

    # remove dashes from the given code
    code = ''.join(groups)

    # check whether or all characters (except the final one) are digits
    if not code[:-1].isdigit():
        return False

    # check the check digit of the given code
    return checkdigit(code) == code[-1]

def checkdigit(code):

    """
    >>> checkdigit('997150210')
    '0'
    >>> checkdigit('938389293')
    '5'
    """

    # compute the check digit
    check = sum((i + 1) * int(code[i]) for i in range(9)) % 11

    # convert the check digit into its string representation
    return 'X' if check == 10 else str(check)

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

6.2 Complementary sequences

```
def increasing(sequence):

    """
    >>> increasing([2, 3, 5, 7, 11, 13])
    True
    >>> increasing((0, 0, 1, 2, 2, 3, 3, 4, 4, 4, 4, 5, 5, 6))
    True
```

```
    >>> increasing([5, 3, 2, 7, 8, 1, 9])
    False
    """

    # check for each pair of successive numbers whether the first number is not
    # larger than the second number
    return all(sequence[i] <= sequence[i + 1] for i in range(len(sequence) - 1))

def frequencySequence(sequence):

    """
    >>> frequencySequence([2, 3, 5, 7, 11, 13])
    [0, 0, 1, 2, 2, 3, 3, 4, 4, 4, 4, 5, 5, 6]
    >>> frequencySequence((0, 0, 1, 2, 2, 3, 3, 4, 4, 4, 4, 5, 5, 6))
    [2, 3, 5, 7, 11, 13, 14]
    >>> frequencySequence([5, 3, 2, 7, 8, 1, 9])
    Traceback (most recent call last):
    AssertionError: given sequence is not increasing
    """

    # check if given sequence is increasing
    assert increasing(sequence), 'given sequence is not increasing'

    freq, value, count = [], 0, 0
    for number in sequence:

        # right now, count indicates how many numbers in the sequence are less
        # than the current number in the sequence
        while value < number:
            freq.append(count)
            value += 1

        count += 1

    # we still have to indicate how many numbers in the sequence are less then
    # or equal to the last number in the sequence (this is all numbers in the
    # sequence)
    freq.append(count)

    # return the frequency sequence
    return freq

def lift(sequence):

    """
    >>> lift([2, 3, 5, 7, 11, 13])
    [3, 5, 8, 11, 16, 19]
    >>> lift((0, 0, 1, 2, 2, 3, 3, 4, 4, 4, 4, 5, 5, 6))
    [1, 2, 4, 6, 7, 9, 10, 12, 13, 14, 15, 17, 18, 20]
    >>> lift([5, 3, 2, 7, 8, 1, 9])
    [6, 5, 5, 11, 13, 7, 16]
    """

    # increase elements in sequence according to their position
    return [element + position + 1 for position, element in enumerate(sequence)]

def complementarySequences(sequence):

    """
    >>> complementarySequences([2, 3, 5, 7, 11, 13])
    ([3, 5, 8, 11, 16, 19], [1, 2, 4, 6, 7, 9, 10, 12, 13, 14, 15, 17, 18, 20])
    >>> complementarySequences((1, 3, 3, 5, 5, 5, 7, 7, 7, 7))
    ([2, 5, 6, 9, 10, 11, 14, 15, 16, 17], [1, 3, 4, 7, 8, 12, 13, 18])
    >>> complementarySequences([5, 3, 2, 7, 8, 1, 9])
    Traceback (most recent call last):
    AssertionError: given sequence is not increasing
    """

    return lift(sequence), lift(frequencySequence(sequence))
```

利用while化简代码

```
if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

6.3 Zipper method

```
def merge(sequence1, sequence2):

    """
    >>> merge(('A', 'B', 'C'),  [1, 2, 3])
    ['A', 1, 'B', 2, 'C', 3]
    >>> merge(['A'], [1, 2, 3, 4])
    ['A', 1]
    >>> merge(('A', 'B'),  (1, 2, 3, 4))
    ['A', 1, 'B', 2]
    >>> merge(('A', 'B', 'C'),  [1, 2])
    ['A', 1, 'B', 2]
    """

    # create new empty list
    merged = []

    # process pairs of elements until shortest sequence is exhausted
    for element1, element2 in zip(sequence1, sequence2):

        # add next pair of elements to the list
        merged.extend((element1, element2))

    # return the merged list
    return merged

def weave(sequence1, sequence2):

    """
    >>> weave(('A', 'B', 'C'),  [1, 2, 3])
    ['A', 1, 'B', 2, 'C', 3]
    >>> weave(['A'], [1, 2, 3, 4])
    ['A', 1, 'A', 2, 'A', 3, 'A', 4]
    >>> weave(('A', 'B'),  (1, 2, 3, 4))
    ['A', 1, 'B', 2, 'A', 3, 'B', 4]
    >>> weave(('A', 'B', 'C'),  [1, 2])
    ['A', 1, 'B', 2, 'C', 1]
    """

    # create new empty list
    woven = []

    # process pairs of elements until shortest sequence is exhausted
    for index in range(max(len(sequence1), len(sequence2))):

        # add next pair of elements to the list; use modulo operator to go back
        # to the start of the sequence each time the end of the sequence is
        # reached
        woven.extend((
            sequence1[index % len(sequence1)],
            sequence2[index % len(sequence2)]
        ))

    # return the woven list
    return woven

def zipper(sequence1, sequence2):

    """
    >>> zipper(('A', 'B', 'C'),  [1, 2, 3])
    ['A', 1, 'B', 2, 'C', 3]
```

```
    >>> zipper(['A'], [1, 2, 3, 4])
    ['A', 1, 2, 3, 4]
    >>> zipper(('A', 'B'),  (1, 2, 3, 4))
    ['A', 1, 'B', 2, 3, 4]
    >>> zipper(('A', 'B', 'C'),  [1, 2])
    ['A', 1, 'B', 2, 'C']
    """

    # determine shortest and longest sequence
    short = sequence1 if len(sequence1) < len(sequence2) else sequence2
    long = sequence1 if len(sequence1) >= len(sequence2) else sequence2

    # create new empty list
    zipped = []

    # process pairs of elements until shortest sequence is exhausted
    for index in range(len(short)):

        # add next pair of elements to the list
        zipped.extend((sequence1[index], sequence2[index]))

    # append additional elements of longest sequence
    zipped.extend(long[len(short):])

    # return the zipped list
    return zipped

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

6.4 Diffy

```
def next(numbers):

    """
    >>> next([32, 9, 14, 3])
    (23, 5, 11, 29)
    >>> next((1, 2, 1, 2, 1, 0))
    (1, 1, 1, 1, 1, 1)
    >>> next((1, 2, 1, 2, 1, 1))
    (1, 1, 1, 1, 0, 0)
    """

    # determine length of the given number sequence
    n = len(numbers)

    # determine next number sequence from the Ducci sequence
    return tuple([abs(numbers[i] - numbers[(i + 1) % n]) for i in range(n)])

def ducci(numbers):

    """
    >>> ducci([32, 9, 14, 3])
    ((32, 9, 14, 3), (23, 5, 11, 29), (18, 6, 18, 6), (12, 12, 12, 12), (0, 0, 0, 0))
    >>> ducci((1, 2, 1, 2, 1, 0))
    ((1, 2, 1, 2, 1, 0), (1, 1, 1, 1, 1, 1), (0, 0, 0, 0, 0, 0))
    >>> ducci((1, 2, 1, 2, 1, 1))
    ((1, 2, 1, 2, 1, 1), (1, 1, 1, 1, 0, 0), (0, 0, 0, 1, 0, 1), (0, 0, 1, 1, 1, 1), (0, 1, 0,
         0, 0, 1), (1, 1, 0, 0, 1, 1), (0, 1, 0, 1, 0, 0), (1, 1, 1, 1, 0, 0))
    """

    # convert given number sequence into a tuple
    numbers = tuple(numbers)

    # initialize Ducci sequence as empty list and empty set; set is used for
    # fast lookup to see if a number sequence is already in the Ducci sequence
```

```
    ducci_list = []
    ducci_set = set()

    # determine possible endpoint of the Ducci sequence as a tuple of zeros
    endpoint = (0, ) * len(numbers)

    # extend Ducci sequence until sequence ends with a tuple of zeros or with a
    # tuple that was already in the sequence (periodic Ducci sequence)
    while numbers != endpoint and numbers not in ducci_set:

        # add next tuple of numbers to the Ducci sequence
        ducci_list.append(numbers)
        ducci_set.add(numbers)

        # determine next tuple of numbers from the Ducci sequence
        numbers = next(numbers)

    # add last tuple of numbers to the Ducci sequence
    ducci_list.append(numbers)

    # convert list containing Ducci sequence into a tuple
    return tuple(ducci_list)

def period(numbers):

    """
    >>> period([32, 9, 14, 3])
    0
    >>> period((1, 2, 1, 2, 1, 0))
    0
    >>> period((1, 2, 1, 2, 1, 1))
    6
    """

    # determine Ducci sequence
    numbers = ducci(numbers)

    # determine period of the Ducci sequence
    return len(numbers) - 1 - numbers.index(numbers[-1])

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

6.5 A square triangle

```
from operator import mul
from functools import reduce

def triangle(rows):

    """
    >>> triangle(0)
    []
    >>> triangle(1)
    [[1]]
    >>> triangle(2)
    [[1], [1, 1]]
    >>> triangle(3)
    [[1], [1, 1], [1, 2, 1]]
    >>> triangle(4)
    [[1], [1, 1], [1, 2, 1], [1, 3, 3, 1]]
    >>> triangle(5)
    [[1], [1, 1], [1, 2, 1], [1, 3, 3, 1], [1, 4, 6, 4, 1]]
    >>> triangle(-1)
    Traceback (most recent call last):
    AssertionError: invalid number of rows
```

```
    >>> triangle(3.14)
    Traceback (most recent call last):
    AssertionError: invalid number of rows
    """

    # number of rows must be a non-negative integer
    assert (isinstance(rows, int) and rows >= 0), 'invalid number of rows'

    # exception: triangle without rows is represented as empty list
    if rows == 0:
        return []

    # initialise triangle with a single row
    triangle = [[1]]

    # each time compute the next row based on the previous row, and append it to
    # the bottom of the triangle (so that it becomes the new previous row)
    for _ in range(1, rows):
        triangle.append(
            [1] +
            [sum(pair) for pair in zip(triangle[-1], triangle[-1][1:])] +
            [1]
        )

    # return triangle with requested number of rows
    return triangle

def hexagon(row, col):

    """
    >>> hexagon(8, 4)
    [15, 20, 35, 70, 56, 21]
    >>> hexagon(16, 7)
    [2002, 3003, 6435, 11440, 8008, 3003]
    >>> hexagon(3, 3)
    Traceback (most recent call last):
    AssertionError: invalid internal position
    """

    # check if given position is an internal cel of the triangle
    assert (
        isinstance(row, int) and
        isinstance(col, int) and
        row > 2 and 1 < col < row
    ), 'invalid internal position'

    # determine Pascal's triangle with one extra row that contains the two
    # neighbouring cells below the given cell
    pascal = triangle(row + 1)

    # return a list containing the numbers in the six neighbouring cells
    return [
        pascal[-r][col - k]
        for r, k in ((3, 2), (3, 1), (2, 0), (1, 0), (1, 1), (2, 2))
    ]

def square(row, col):

    """
    >>> square(8, 4)
    '15 x 20 x 35 x 70 x 56 x 21 = 864360000 = 29400 x 29400'
    >>> square(16, 7)
    '2002 x 3003 x 6435 x 11440 x 8008 x 3003 = 10643228293383247161600 = 103166022960 x
        103166022960'
    >>> square(3, 3)
    Traceback (most recent call last):
    AssertionError: invalid internal position
    """
```

```
    # determine numbers in six neighbouring cells of the given cell
    sequence = hexagon(row, col)

    # compute product of six neighbouring numbers
    square = reduce(mul, sequence, 1)

    # output factors, product and square root
    return '{0} = {1} = {2} x {2}'.format(
        ' x '.join(str(n) for n in sequence),
        square,
        round(square ** 0.5)
    )

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

6.6 Pozo Azul

```
def crossSection(rows, passages):

    """
    >>> crossSection(4, 'NSSWNSSWNWNWEWSWNSSEEWSWEWSENSSWNENWNSNEEWEWSWSENWNESEEWNWNWSESW')
    [['NS', 'SW', 'NS', 'SW', 'NW', 'NW', 'EW', 'SW'], ['NS', 'SE', 'EW', 'SW', 'EW', 'SE', '
        NS', 'SW'], ['NE', 'NW', 'NS', 'NE', 'EW', 'EW', 'SW', 'SE'], ['NW', 'NE', 'SE', 'EW',
         'NW', 'NW', 'SE', 'SW']]
    >>> crossSection(4, 'NSSWNSSWNWNWEWSWNSS')
    Traceback (most recent call last):
    AssertionError: invalid cross section
    """

    # check if given string can be used to construct a rectangular grid
    assert len(passages) % (2 * rows) == 0, 'invalid cross section'

    # construct case with passages in cross section
    cols = len(passages) // (2 * rows)
    return [
        [
            passages[2 * (row * cols + col):2 * (row * cols + col + 1)]
            for col in range(cols)
        ]
        for row in range(rows)
    ]

def depth(cave):

    """
    >>> cave = crossSection(4, '
        NSSWNSSWNWNWEWSWNSSEEWSWEWSENSSWNENWNSNEEWEWSWSENWNESEEWNWNWSESW')
    >>> depth(cave)
    11
    """

    # determine number of rows and columns of the given cave
    rows, cols = len(cave), len(cave[0])

    # determine possible movements in a cave
    movements = {
        'N': (-1, 0, 'S'),
        'S': (1, 0, 'N'),
        'E': (0, 1, 'W'),
        'W': (0, -1, 'E')
    }

    # define starting position and initial depth
    r, c, depth = -1, 0, 0
```

```
    # define initial movement
    dr, dc, ri = movements['S']

    while (
        0 <= r + dr < rows and
        0 <= c + dc < cols and
        ri in cave[r + dr][c + dc]
    ):

        # we go one step deeper into the cave
        depth += 1

        # take a step to the neighbouring cell
        r += dr
        c += dc

        # determine direction in which we will leave the neighbouring cell
        dr, dc, ri = movements[cave[r][c].replace(ri, '')]

    return depth

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

7 Series 07: advanced functions and modules

7.1 ISBN

```
def isISBN10(code):

    """
    Checks whether or not the given ISBN-10 code is valid.

    >>> isISBN10('9971502100')
    True
    >>> isISBN10('9971502108')
    False
    """

    # helper function for computing the ISBN-10 check digit
    def checkdigit(code):

        # compute the check digit
        check = sum((i + 1) * int(code[i]) for i in range(9)) % 11

        # convert the check digit into string representation
        return 'X' if check == 10 else str(check)

    # check whether the given code is a string
    if not isinstance(code, str):
        return False

    # check whether the given code contains 10 characters
    if len(code) != 10:
        return False

    # check whether first nine characters of the given code are digits
    if not code[:9].isdigit():
        return False

    # check the check digit
    return checkdigit(code) == code[-1]

def isISBN13(code):

    """
    Checks whether or not the given ISBN-13 code is valid.

    >>> isISBN13('9789743159664')
    True
    >>> isISBN13('9787954527409')
    False
    >>> isISBN13('8799743159665')
    False
    """

    # helper function for computing the ISBN-10 check digit
    def checkdigit(code):

        # compute the check digit
        check = sum((3 if i % 2 else 1) * int(code[i]) for i in range(12))

        # convert the check digit into a single digit
        return str((10 - check) % 10)

    # check whether the given code is a string
    if not isinstance(code, str):
        return False

    # check whether the given code contains 10 characters
    if len(code) != 13:
        return False
```

```
    # check whether first nine characters of the given code are digits
    if not code[:12].isdigit():
        return False

    # check the check digit
    return checkdigit(code) == code[-1]

def isISBN(code, isbn13=True):

    """
    >>> isISBN('9789027439642', False)
    False
    >>> isISBN('9789027439642', True)
    True
    >>> isISBN('9789027439642')
    True
    >>> isISBN('080442957X')
    False
    >>> isISBN('080442957X', False)
    True
    """

    return isISBN13(code) if isbn13 else isISBN10(code)

def areISBN(codes, isbn13=None):

    """
    >>> areISBN(
    ...     [
    ...         '0012345678', '0012345679', '9971502100', '080442957X',
    ...         5, True, 'The Practice of Computing Using Python',
    ...         '9789027439642', '5486948320146'
    ...     ]
    ... )
    [False, True, True, True, False, False, False, True, False]

    >>> areISBN(
    ...     [
    ...         '0012345678', '0012345679', '9971502100', '080442957X',
    ...         5, True, 'The Practice of Computing Using Python',
    ...         '9789027439642', '5486948320146'
    ...     ],
    ...     True
    ... )
    [False, False, False, False, False, False, False, True, False]

    >>> areISBN(
    ...     [
    ...         '0012345678', '0012345679', '9971502100', '080442957X',
    ...         5, True, 'The Practice of Computing Using Python',
    ...         '9789027439642', '5486948320146'
    ...     ],
    ...     False
    ... )
    [False, True, True, True, False, False, False, False, False]
    """

    # initialize list of evaluations
    evaluations = []

    # construct list of evaluations
    for code in codes:
        if isinstance(code, str):
            if isbn13 is None:
                if len(code) == 13:
                    evaluations.append(isISBN(code, True))
                else:
                    evaluations.append(isISBN(code, False))
```

```
            else:
                evaluations.append(isISBN(code, isbn13))
        else:
            evaluations.append(False)

    # return list of results
    return evaluations

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

7.2 Baseball

```
def hit(base, occupied=None):

    """
    >>> hit(2)
    (0, [2])
    >>> hit(0, [1, 3])
    (0, [1, 3])
    >>> hit(1, (1, 3))
    (1, [1, 2])
    >>> hit(2, occupied=[1, 3])
    (1, [2, 3])
    >>> hit(3, occupied=(1, 3))
    (2, [3])
    >>> hit(4, occupied=[1, 3])
    (3, [])
    """

    # by default, no bases are occupied before the hit
    if occupied is None:
        occupied = []

    # move players that occupy bases forward based on hit
    occupied = [prev_base + base for prev_base in occupied]

    # bring batter into the field
    if base:
        occupied.append(base)

    # determine score
    score = len([base for base in occupied if base >= 4])

    # determine new occupation of bases
    occupied = sorted([base for base in occupied if base < 4])

    # return score en new occupation of bases
    return score, occupied

def inning(bases):

    """
    >>> inning([0, 1, 2, 3, 4])
    (4, [])
    >>> inning((4, 3, 2, 1, 0))
    (2, [1, 3])
    >>> inning([1, 1, 2, 1, 0, 0, 1, 3, 0])
    (5, [3])
    """

    # no score and no occupied bases at the start of the inning
    total, occupied = 0, []

    # simulate hits during the inning
    for base in bases:
```

```
        # determine score and new base occupation after next hit
        score, occupied = hit(base, occupied)
        total += score

    return total, occupied

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

7.3 Rollover calendar

```
from datetime import date, timedelta

def rolloverDate(day=None, month=None, year=None):

    """
    >>> rolloverDate(31, 4)
    datetime.date(2016, 5, 1)
    >>> rolloverDate(43, 15, 2016)
    datetime.date(2017, 4, 12)
    >>> rolloverDate(year=2016, month=3, day=16)
    datetime.date(2016, 3, 16)
    >>> rolloverDate(year=2016, month=12, day=64)
    datetime.date(2017, 2, 2)
    >>> rolloverDate(year=2016, month=19, day=99)
    datetime.date(2017, 10, 7)
    >>> rolloverDate(year=2016, month=1, day=99999)
    datetime.date(2289, 10, 14)
    >>> rolloverDate(year=2016, month=9999, day=10)
    datetime.date(2849, 3, 10)
    """

    # determine today's date if a default value is needed
    if year is None or month is None or day is None:
        today = date.today()

    # deterime year, month, day from given values and default values
    year = year if year is not None else today.year
    month = month if month is not None else today.month
    day = day if day is not None else today.day

    # rollover the number of months
    year += (month - 1) // 12
    month = 1 + (month - 1) % 12

    # rollover the number of days
    return date(year, month, 1) + timedelta(day - 1)

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

7.4 The billion-year war

```
def reverseComplement(sequence):

    """
    >>> reverseComplement('GATATC')
    'GATATC'
    >>> reverseComplement('GCATGC')
    'GCATGC'
    >>> reverseComplement('AGCTTC')
```

```
    'GAAGCT'
    """

    # dictionary that maps each base onto its complementary base
    complement = {b1:b2 for b1, b2 in zip('ACGT', 'TGCA')}

    # invert and complement the given sequence
    return ''.join(complement[base] for base in sequence[::-1])

def reversePalindrome(sequence):

    """
    >>> reversePalindrome('GATATC')
    True
    >>> reversePalindrome('GCATGC')
    True
    >>> reversePalindrome('AGCTTC')
    False
    """

    # check if sequence is equal to its inverse complement
    return sequence == reverseComplement(sequence)

def restrictionSites(sequence, minLength=4, maxLength=12):

    """
    >>> restrictionSites('TCAATGCATGCGGGTCTATATGCAT')
    [(4, 'ATGCAT'), (5, 'TGCA'), (6, 'GCATGC'), (7, 'CATG'), (17, 'TATA'), (18, 'ATAT'), (20,
        'ATGCAT'), (21, 'TGCA')]
    >>> restrictionSites('AAGTCATAGCTATCGATCAGATCAC', minLength=5)
    [(6, 'ATAGCTAT'), (7, 'TAGCTA'), (12, 'ATCGAT')]
    >>> restrictionSites('ATATTCAGTCATCGATCAGCTAGCA', maxLength=5)
    [(1, 'ATAT'), (12, 'TCGA'), (14, 'GATC'), (18, 'AGCT'), (20, 'CTAG')]
    """

    # loop over all possible subsequences (taking into account the minimal and
    # maximal length) and check whether they are palindromes; subsequences are
    # traversed in the order in which the palindromes need to occur in the list,
    # avoiding the need for an extra sorting step
    sites = []
    for start in range(len(sequence) - minLength+1):
        length = minLength
        while length <= maxLength and start + length <= len(sequence):
            subsequence = sequence[start:start + length]
            if reversePalindrome(subsequence):
                sites.append((start + 1, subsequence))
            length += 1

    # return the sorted list of restriction sites
    return sites

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

7.5 Cool serial numbers

```
def serialNumber(number):

    """
    >>> serialNumber(834783)
    '00834783'
    >>> serialNumber('47839')
    '00047839'
    >>> serialNumber(834783244839184)
    '834783244839184'
    >>> serialNumber('4783926132432*')
```

```
    Traceback (most recent call last):
    AssertionError: invalid serial number
    """

    assert (
        (isinstance(number, int) and number > 0) or
        (isinstance(number, str) and number.isdigit() and int(number) != 0)
    ), 'invalid serial number'

    return str(number).zfill(8)

def solid(number):

    """
    In a solid serial number, every digit is the same.

    >>> solid(44444444)
    True
    >>> solid('44544444')
    False
    """

    number = serialNumber(number)
    return number == number[0] * len(number)

def radar(number):

    """
    In a radar serial number, the serial number reads the same left-to-right as
    it does right-to-left.

    >>> radar(1133110)
    True
    >>> radar('83289439')
    False
    """

    number = serialNumber(number)
    half = len(number) // 2
    return number[:half] == number[half:][::-1] and not solid(number)

def repeater(number):

    """
    In a repeater serial number, the second half of the serial number is the
    same as the first half.

    >>> repeater(20012001)
    True
    >>> repeater('83289439')
    False
    """

    number = serialNumber(number)
    half = len(number) // 2
    return number[:half] == number[half:] and not solid(number)

def radarRepeater(number):

    """
    A radar repeater is both a radar and a repeater.

    >>> radarRepeater('12211221')
    True
    >>> radarRepeater('83289439')
    False
    """

    return radar(number) and repeater(number)
```

```
def numismatist(series, kind=solid):

    """
    >>> numismatist([33333333, 1133110, '77777777', '12211221'])
    [33333333, '77777777']
    >>> numismatist([33333333, 1133110, '77777777', '12211221'], radar)
    [1133110, '12211221']
    >>> numismatist([33333333, 1133110, '77777777', '12211221'], kind=repeater)
    ['12211221']
    """

    return [number for number in series if kind(number)]

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

7.6 Error detection

```
from random import choice, randint

def draw(drawn=None):

    """
    >>> draw()
    '6S'
    >>> draw(['6H', '3C', '3D', '8C', 'AD', '9D', '7D', 'QC'])
    '4D'
    >>> draw(drawn=('3S', '8H', '8C', '2H', 'AC'))
    'XH'
    >>> draw({'4C', 'AH', 'JS', '7S', '9H', '2H', 'QC', '2S', '3H', '7C'})
    '9S'
    """

    # make a complete deck of cards
    deck = {rank + suit for rank in 'A23456789XJQK' for suit in 'SHCD'}

    # remove cards that have already been drawn from the deck (if any)
    if drawn is not None:
        deck -= set(drawn)

    # draw a random card and return it
    return choice(list(deck))

    """
    alternative solution (less efficient if many cards were already drawn)

    # define suits and ranks of the cards
    ranks = 'A23456789XJQK'
    suits = 'SHDC'

    # draw a random card
    card = choice(ranks) + choice(suits)

    # keep drawing random cards until one is found that hasn't been drawn yet
    while drawn is not None and card in drawn:
        card = choice(ranks) + choice(suits)

    # return randomly drawn card
    return card
    """

def arrange(rows=5, cols=5):

    """
    >>> arrange(rows=3, cols=4)
```

```
    [['5D', '4D', '4C', '9S'], ['2D', '6C', '4S', 'AD'], ['QH', 'QS', '2S', '3D']]
    >>> arrange(rows=7, cols=8)
    Traceback (most recent call last):
    AssertionError: invalid grid
    """

    # check validity of grid
    assert (
        rows >= 1 and           # at least one row
        cols >= 1 and           # at least one column
        rows * cols <= 52       # not more than 52 cards in grid
    ), 'invalid grid'

    # initially no cards are drawn
    drawn = set()

    # construct the grid
    grid = []
    for _ in range(rows):
        row = []
        for _ in range(cols):
            card = draw(drawn)
            row.append(card)
            drawn.add(card)
        grid.append(row)

    return grid

def extend(grid):

    """
    >>> grid = [['QH', '9S', '3C'], ['5D', '8C', '2H']]
    >>> extend(grid)
    >>> grid
    [['QH', '9S', '3C', 'JH'], ['5D', '8C', '2H', '9H'], ['XD', 'XC', '4C', '9C']]
    """

    # check validity of grid
    assert (
        len(grid) >= 1 and      # at least one row
        len(grid[0]) >= 1 and   # at least one column
        # after extension not more than 52 cards
        (len(grid) + 1) * (len(grid[0]) + 1) <= 52
    ), 'invalid grid'

    # determine which cards have already been arranged in the grid
    drawn = set()
    for row in grid:
        drawn.update(row)

    # add an extra card at  the end of each row
    for row in grid:
        card = draw(drawn)
        row.append(card)
        drawn.add(card)

    # add an extra row of cards
    row = []
    for _ in range(len(grid[0])):
        card = draw(drawn)
        row.append(card)
        drawn.add(card)
    grid.append(row)

def select(grid):

    """
    >>> grid = [['RA', 'K6', 'RV', 'H7'], ['R6', 'KX', 'KX', 'KV'], ['R8', 'R4', 'R7', 'K3']]
    >>> select(grid)
```

```
    (1, 3)
    """

    # select a random position in the grid
    return randint(0, len(grid) - 1), randint(0, len(grid[0]) - 1)

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

8 Series 08: sets and dictionaries

8.1 ISBN

```
def isISBN13(code):

    """
    Checks whether or not the given ISBN-13 code is valid.

    >>> isISBN13('9789743159664')
    True
    >>> isISBN13('9787954527409')
    False
    >>> isISBN13('8799743159665')
    False
    """

    def checkdigit(code):

        """
        Helper function that computes the ISBN-13 check digit.
        """

        # compute the check digit
        check = sum((2 * (i % 2) + 1) * int(code[i]) for i in range(12))

        # convert the check digit into a single digit
        return str((10 - check) % 10)

    # check whether the given code is a string
    if not isinstance(code, str):
        return False

    # check whether the given code contains 10 characters
    if len(code) != 13:
        return False

    # check prefix of the given code
    if code[:3] not in {'978', '979'}:
        return False

    # check whether first nine characters of the given code are digits
    if not code[:12].isdigit():
        return False

    # check the check digit
    return checkdigit(code) == code[-1]

def overview(codes):

    """
    >>> codes = [
    ...     '9789743159664', '9785301556616', '9797668174969', '9781787559554',
    ...     '9780817481461', '9785130738708', '9798810365062', '9795345206033',
    ...     '9792361848797', '9785197570819', '9786922535370', '9791978044523',
    ...     '9796357284378', '9792982208529', '9793509549576', '9787954527409',
    ...     '9797566046955', '9785239955499', '9787769276051', '9789910855708',
    ...     '9783807934891', '9788337967876', '9786509441823', '9795400240705',
    ...     '9787509152157', '9791478081103', '9780488170969', '9795755809220',
    ...     '9793546666847', '9792322242176', '9782582638543', '9795919445653',
    ...     '9796783939729', '9782384928398', '9787590220100', '9797422143460',
    ...     '9798853923096', '9784177414990', '9799562126426', '9794732912038',
    ...     '9787184435972', '9794455619207', '9794270312172', '9783811648340',
    ...     '9799376073039', '9798552650309', '9798485624965', '9780734764010',
    ...     '9783635963865', '9783246924279', '9797449285853', '9781631746260',
    ...     '9791853742292', '9781796458336', '9791260591924', '9789367398012'
    ... ]
    >>> overview(codes)
```

```
    English speaking countries: 8
    French speaking countries: 4
    German speaking countries: 6
    Japan: 3
    Russian speaking countries: 7
    China: 8
    Other countries: 11
    Errors: 9
    """

    # construct histogram of registration groups
    groups = {}
    for i in range(11):
        groups[i] = 0
    for code in codes:
        if not isISBN13(code):
            groups[10] += 1
        else:
            groups[int(code[3])] += 1

    # display overview
    print('English speaking countries: {}'.format(groups[0] + groups[1]))
    print('French speaking countries: {}'.format(groups[2]))
    print('German speaking countries: {}'.format(groups[3]))
    print('Japan: {}'.format(groups[4]))
    print('Russian speaking countries: {}'.format(groups[5]))
    print('China: {}'.format(groups[7]))
    print('Other countries: {}'.format(groups[6] + groups[8] + groups[9]))
    print('Errors: {}'.format(groups[10]))

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

8.2 Runs and groups

```
def three_different(tiles):

    """
    >>> three_different(['4R', '4B', '4Y', '4K'])
    True
    >>> three_different({'6B', '7B', '8B', '9B', '10B'})
    True
    >>> three_different(('11R', '2B', '7Y', '2B', '9K'))
    False
    """

    # check if there are at least three tiles and if all tiles are different
    return (
        len(tiles) >= 3 and            # at least three tiles
        len(tiles) == len(set(tiles))  # all tiles are different
    )

def group(tiles):

    """
    >>> group(['4R', '4B', '4Y', '4K'])
    True
    >>> group({'6B', '7B', '8B', '9B', '10B'})
    False
    >>> group(('11R', '2B', '7Y', '2B', '9K'))
    False
    """

    # check if there are at least three tiles that are all different
    if not three_different(tiles):
        return False
```

```
    # check if all tiles have a distinct color
    colors = {tile[-1] for tile in tiles}
    if len(colors) != len(tiles):
        return False

    # check if all tiles have the same value
    values = {tile[:-1] for tile in tiles}
    if len(values) != 1:
        return False

    # all conditions for a group of tiles are fulfilled
    return True

def run(tiles):

    """
    >>> run(['4R', '4B', '4Y', '4K'])
    False
    >>> run({'6B', '7B', '8B', '9B', '10B'})
    True
    >>> run(('11R', '2B', '7Y', '2B', '9K'))
    False
    """

    # check if there are at least three tiles that are all different
    if not three_different(tiles):
        return False

    # check if all tiles have the same color
    colors = {tile[-1] for tile in tiles}
    if len(colors) != 1:
        return False

    # check if ascending tile values form consecutive sequence of integers
    values = sorted(int(tile[:-1]) for tile in tiles)
    for i in range(len(values) - 1):
        if values[i + 1] - values[i] != 1:
            return False

    # all conditions for a run of tiles are fulfilled
    return True

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

8.3 Changing gender

```
def translate(word, translations):

    """
    >>> translations = {'he':'she', 'brother':'sister'}
    >>> translate('he', translations)
    'she'
    >>> translate('HE', translations)
    'SHE'
    >>> translate('He', translations)
    'She'
    >>> translate('brother', translations)
    'sister'
    >>> translate('my', translations)
    'my'
    """

    # word is only translated if its lowercase version occurs in the given
    # dictionary
```

```
    if word.lower() in translations:

        # lookup translation of word (with lowercase variant)
        translation = translations[word.lower()]

        # mimick use of case
        if word.isupper():
            translation = translation.upper()
        elif word == word.capitalize():
            translation = translation.capitalize()

        # use translation as new version of word
        word = translation

    # return original word or its translation
    return word

def sexChange(sentence, translations):

    """
    >>> translations = {'he':'she', 'brother':'sister'}
    >>> sexChange('He is my brother.', translations)
    'She is my sister.'
    """

    # split sentence into words and apply translation on each word
    word, translation = '', ''
    for character in sentence:
        if character.isalpha():
            word += character
        else:
            translation += translate(word, translations) + character
            word = ''

    # return translated sentence
    return translation + translate(word, translations)

def undoSexChange(sentence, translations):

    """
    >>> translations = {'he':'she', 'brother':'sister'}
    >>> undoSexChange('She is my sister.', translations)
    'He is my brother.'
    """

    # apply reverse translation on each word of the given sentence
    return sexChange(
        sentence,
        {translation:word for word, translation in translations.items()}
    )

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

8.4 Sacred Cat of Burma

```
def color(genotype):

    """
    >>> color('CcDd')
    'seal'
    >>> color('ccdd')
    'lilac'
    """

    if 'C' in genotype:
```

```
        return 'seal' if 'D' in genotype else 'blue'
    else:
        return 'chocolate' if 'D' in genotype else 'lilac'

def combinations(genotype):

    """
    >>> combinations('CcDd')
    ['CD', 'Cd', 'cD', 'cd']
    >>> combinations('ccdd')
    ['cd', 'cd', 'cd', 'cd']
    """

    # list containing four possible combinations (in generic order)
    return [c + d for c in genotype[:2] for d in genotype[2:]]

def punnett(father, mother, pprint=False):

    """
    >>> print(punnett('CcDd', 'CcDd'))
    [['CCDD', 'CCDd', 'CcDD', 'CcDd'], ['CCdD', 'CCdd', 'CcdD', 'Ccdd'], ['cCDD', 'cCDd', '
        ccDD', 'ccDd'], ['cCdD', 'cCdd', 'ccdD', 'ccdd']]
    >>> print(punnett('CcDd', 'CcDd', pprint=True))
    CCDD CCDd CcDD CcDd
    CCdD CCdd CcdD Ccdd
    cCDD cCDd ccDD ccDd
    cCdD cCdd ccdD ccdd
    >>> print(punnett('cCDd', 'CcdD', pprint=True))
    cCDd cCDD ccDd ccDD
    cCdd cCdD ccdd ccdD
    CCDd CCDD CcDd CcDD
    CCdd CCdD Ccdd CcdD
    """

    # generate Punnett square (as a string or a nested list)
    if pprint:
        return '\n'.join(
            ' '.join(v[0] + m[0] + v[1] + m[1] for m in combinations(mother))
            for v in combinations(father)
        )
    else:
        return [
            [v[0] + m[0] + v[1] + m[1] for m in combinations(mother)]
            for v in combinations(father)
        ]

def colorDistribution(father, mother):

    """
    >>> colorDistribution('CcDd', 'CcDd') == {'blue': 3, 'seal': 9, 'lilac': 1, 'chocolate':
        3}
    True
    >>> colorDistribution('cCDD', 'cCDD') == {'seal': 12, 'chocolate': 4}
    True
    >>> colorDistribution('ccDD', 'ccDD')
    {'chocolate': 16}
    >>> colorDistribution('ccdd', 'CcDd') == {'blue': 4, 'lilac': 4, 'seal': 4, 'chocolate':
        4}
    True
    >>> colorDistribution('ccdd', 'CCDD')
    {'seal': 16}
    >>> colorDistribution('ccdd', 'CcDD') == {'chocolate': 8, 'seal': 8}
    True
    >>> colorDistribution('ccdd', 'ccDD')
    {'chocolate': 16}
    >>> colorDistribution('ccdd', 'ccDd') == {'lilac': 8, 'chocolate': 8}
    True
    >>> colorDistribution('ccdd', 'Ccdd') == {'blue': 8, 'lilac': 8}
    True
```

```
    >>> colorDistribution('ccdd', 'CCdd')
    {'blue': 16}
    >>> colorDistribution('ccdd', 'ccdd')
    {'lilac': 16}
    """

    # loop over all genotypes in the Punnett square of the given father and
    # mother and increment the counter of the corresponding point colour
    distribution = {}
    for row in punnett(father, mother):
        for genotype in row:
            c = color(genotype)
            distribution[c] = distribution.get(c, 0) + 1

    # return distribution of the point colours of the offspring
    return distribution

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

8.5 What's in the bag?

```
def fill(letters):

    """
    >>> bag = fill('
        IAMDIETINGIEATQUINCEJELLYLOTSOFGROUNDMAIZEGIVESVARIETYICOOKRHUBARBANDSODAWEEPANEWORPUTONEXTRAFLESH_
        ')
    >>> bag
    {'U': 4, '_': 2, 'C': 2, 'K': 1, 'D': 4, 'T': 6, 'Q': 1, 'V': 2, 'A': 9, 'F': 2, 'O': 8, '
        J': 1, 'I': 9, 'N': 6, 'P': 2, 'S': 4, 'M': 2, 'W': 2, 'E': 12, 'Z': 1, 'G': 3, 'Y':
        2, 'B': 2, 'L': 4, 'R': 6, 'X': 1, 'H': 2}
    >>> description(bag)
    {1: {'Q', 'Z', 'X', 'K', 'J'}, 2: {'F', '_', 'P', 'C', 'M', 'W', 'Y', 'B', 'V', 'H'}, 3:
        {'G'}, 4: {'U', 'D', 'L', 'S'}, 6: {'N', 'R', 'T'}, 8: {'O'}, 9: {'I', 'A'}, 12: {'E
        '}}
    >>> remove('AEERTYOXMCNB_S', bag)
    >>> description(bag)
    {1: {'J', '_', 'C', 'K', 'M', 'Z', 'Y', 'B', 'Q'}, 2: {'W', 'P', 'V', 'F', 'H'}, 3: {'S',
        'G'}, 4: {'U', 'D', 'L'}, 5: {'N', 'R', 'T'}, 7: {'O'}, 8: {'A'}, 9: {'I'}, 10: {'E'}}
    >>> remove('XXX', bag)
    Traceback (most recent call last):
    AssertionError: not all letters are in the bag
     """

    # represent bag as frequency table of given letters
    bag = {}
    for letter in letters:
        bag[letter] = bag.get(letter, 0) + 1

    # return frequency table
    return bag

def description(bag):

    # reverse dictionary that represents bag, with keys grouped into set
    result = {}
    for letter, count in bag.items():
        if count in result:
            result[count].add(letter)
        else:
            result[count] = {letter}

    return result

def remove(letters, bag):
```

```
    # check if all given letters are in the bag
    needed = fill(letters)
    assert all(
        letter in bag and needed[letter] <= bag[letter]
        for letter in needed
    ), 'not all letters are in the bag'

    # remove given letters from the bag
    for letter in letters:
        if bag[letter] == 1:
            del bag[letter]
        else:
            bag[letter] -= 1

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

8.6 Catch as catch can

```
def missing_parameter(given_parameters, equation_parameters):

    """
    >>> missing_parameter({'F':1.2, 'D':0.6, 'H':2, 'B':4}, 'FDVBH')
    'V'
    >>> missing_parameter({'D': 0.6, 'B': 4, 'V': 0.3, 'H': 2}, 'FDVBH')
    'F'
    >>> missing_parameter({'F':1.2, 'D':0.6, 'H':2, 'X':4}, 'FDVBH')
    Traceback (most recent call last):
    AssertionError: invalid parameters
    >>> missing_parameter({'F':1.2, 'D':0.6, 'H':2}, 'FDVBH')
    Traceback (most recent call last):
    AssertionError: invalid parameters
    """

    # check if all given parameters are also parameters used in the formula, and
    # if all-but-one of the equation parameters are also given
    equation_parameters = set(equation_parameters)
    given_parameters = set(given_parameters)
    assert (
        given_parameters < equation_parameters and
        len(equation_parameters - given_parameters) == 1
    ), 'invalid parameters'

    # lookup and return the missing parameter of the equation
    return (equation_parameters - given_parameters).pop()

def juggle(parameters):

    """
    >>> juggle({'F':1.2, 'D':0.6, 'H':2, 'B':4})
    {'F': 1.2, 'D': 0.6, 'B': 4, 'V': 0.3, 'H': 2}
    >>> juggle({'D': 0.6, 'B': 4, 'V': 0.3, 'H': 2})
    {'D': 0.6, 'V': 0.3, 'F': 1.2, 'H': 2, 'B': 4}
    >>> juggle({'F':1.2, 'D':0.6, 'H':2, 'X':4})
    Traceback (most recent call last):
    AssertionError: invalid parameters
    """

    # find missing parameter (AssertionError if invalid parameters given)
    missing = missing_parameter(parameters, 'FDVBH')

    # compute value of missing parameter
    if missing == 'B':
        H, F, D, V = (parameters[key] for key in 'HFDV')
        parameters['B'] = H * (F + D) / (V + D)
```

```
    elif missing == 'H':
        B, V, D, F = (parameters[key] for key in 'BVDF')
        parameters['H'] = B * (V + D) / (F + D)
    elif missing == 'F':
        B, V, D, H = (parameters[key] for key in 'BVDH')
        parameters['F'] = B * (V + D) / H - D
    elif missing == 'V':
        H, F, D, B = (parameters[key] for key in 'HFDB')
        parameters['V'] = H * (F + D) / B - D
    else:
        H, F, B, V = (parameters[key] for key in 'HFBV')
        parameters['D'] = (H * F - B * V) / (B - H)

    # return dictionary with floating point values for all parameterss
    return {key:float(value) for key, value in parameters.items()}

def juggler(**kwargs):

    """
    >>> juggler(F=1.2, D=0.6, H=2, B=4)
    {'F': 1.2, 'D': 0.6, 'B': 4, 'V': 0.3, 'H': 2}
    >>> juggler(D=0.6, B=4, V=0.3, H=2)
    {'D': 0.6, 'V': 0.3, 'F': 1.2, 'H': 2, 'B': 4}
    >>> juggler(F=1.2, D=0.6, H=2, X=4)
    Traceback (most recent call last):
    AssertionError: invalid parameters
    """

    return juggle(kwargs)

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

9 Series 09: text files

9.1 ISBN

```
def isISBN13(code):

    """
    Checks whether or not the given ISBN-13 code is valid.

    >>> isISBN13('9789743159664')
    True
    >>> isISBN13('9787954527409')
    False
    >>> isISBN13('8799743159665')
    False
    """

    def checkdigit(code):

        """
        Helper function that computes the ISBN-13 check digit.
        """

        # compute the check digit
        check = sum((2 * (i % 2) + 1) * int(code[i]) for i in range(12))

        # convert the check digit into a single digit
        return str((10 - check) % 10)

    # check whether the given code is a string
    if not isinstance(code, str):
        return False

    # check whether the given code contains 10 characters
    if len(code) != 13:
        return False

    # check prefix of the given code
    if code[:3] not in {'978', '979'}:
        return False

    # check whether first nine characters of the given code are digits
    if not code[:12].isdigit():
        return False

    # check the check digit
    return checkdigit(code) == code[-1]

def remove_tags(s):

    """
    Removes all XML tags from the given string and then removes all leading and
    trailing whitespace.

    >>> remove_tags(' <Title> The Practice of Computing using <b>Python</b> </Title> ')
    'The Practice of Computing using Python'
    """

    # removes all XML tags from the given string
    s = s.strip()
    while s.find('<') >= 0:
        start = s.find('<')
        stop = s.find('>')
        if stop == -1:
            stop = len(s)
        s = s[:start] + s[stop+1:]

    # removes leading and trailing whitespace and returns the modified string
```

```
    return s.strip()

def displayBookInfo(code):

    """
    >>> displayBookInfo('9780136110675')
    Title: The Practice of Computing using Python
    Authors: William F Punch, Richard Enbody
    Publisher: Addison Wesley
    >>> displayBookInfo('9780136110678')
    Wrong ISBN-13 code
    """

    # check validity of ISBN-13 code
    if not isISBN13(code):
        print('Wrong ISBN-13 code')
        return

    # open web page with URL of ISBNdb.com that provides information about a
    # given ISBN-13 code
    import urllib.request
    url = 'http://isbndb.com/api/books.xml'
    parameters = '?access_key=ZFD8L2Z5&index1=isbn&value1=' + code.strip()
    info = urllib.request.urlopen(url + parameters)

    # extract and output selected book information from XML
    for line in info:
        line = line.decode('utf-8')
        if line.startswith('<Title>'):
            print('Title: {}'.format(remove_tags(line)))
        elif line.startswith('<AuthorsText>'):
            print('Authors: {}'.format(remove_tags(line).rstrip(', ')))
        elif line.startswith('<PublisherText '):
            print('Publisher: {}'.format(remove_tags(line).rstrip(', ')))

    # close web page
    info.close()

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

9.2 Say it like Adele

```
def mix(source1, source2, destination=None):

    """
    Mixes the lines of two source files into the destination file.

    >>> mix('tom_waits.txt', 'adele.txt')
    Operator, number, please
    -->Hello from the other side<--
    It's been so many years
    -->I must have called a thousand times<--
    Will she remember my old voice
    -->To tell you I'm sorry for everything that I've done<--
    While I fight the tears?
    -->But when I call you never seem to be home<--

    >>> mix('tom_waits.txt', 'adele.txt', 'mix.txt')
    >>> print(open('mix.txt', 'r').read(), end='')
    Operator, number, please
    -->Hello from the other side<--
    It's been so many years
    -->I must have called a thousand times<--
    Will she remember my old voice
    -->To tell you I'm sorry for everything that I've done<--
```

```
    While I fight the tears?
    -->But when I call you never seem to be home<--
    """

    # open source for reading
    infile1 = open(source1, 'r')
    infile2 = open(source2, 'r')

    # open destination file for writing; in case the file did not exist yet, a
    # new file is created; otherwise the existing file is overwritten
    outfile = open(destination, 'w') if destination is not None else None

    # continue reading the next line line from each of the two source files, and
    # output them one after the other, with the lines of the second file put in
    # between --> and <-- token
    for line1, line2 in zip(infile1, infile2):
        print(line1, end='', file=outfile)
        print('-->{}<--'.format(line2.rstrip('\n')), file=outfile)

    # explicitely close the source files
    infile1.close()
    infile2.close()

    # explicitely close the destination file if it was opened
    if outfile is not None:
        outfile.close()

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

9.3 AC Melon

```
def pattern(word):

    """
    >>> pattern('AC Melon')
    '_C M_l_n'
    >>> pattern('slipstack')
    'sl_pst_ck'
    >>> pattern('Wander Women')
    'W_nd_r W_m_n'
    """

    # replace all vowels by an underscore
    return ''.join(
        letter if letter.lower() not in 'aeiou' else '_'
        for letter in word
    )

def bloopers(filename, length=1, occurrences=1):

    """
    >>> candidates = bloopers('wheeloffortune.txt')
    >>> candidates['_C M_l_n']
    {'AC Melon', 'AC Milan'}
    >>> candidates['sl_pst_ck']
    {'slapstick', 'slipstack'}
    >>> candidates['W_nd_r W_m_n']
    {'Winder Woman', 'Wander Women', 'Wonder Woman'}

    >>> bloopers('wheeloffortune.txt', length=13)
    {'B_tm_n _nd R_b_n': {'Batman and Robin', 'Batmen and Reban'}}
    >>> bloopers('wheeloffortune.txt', occurrences=3)
    {'W_nd_r W_m_n': {'Wander Women', 'Winder Woman', 'Wonder Woman'}}
    >>> bloopers('wheeloffortune.txt', occurrences=2, length=12)
```

```
    {'W_nd_r W_m_n': {'Wander Women', 'Winder Woman', 'Wonder Woman'}, 'B_tm_n _nd R_b_n': {'
        Batman and Robin', 'Batmen and Reban'}}
    """

    # build dictionary that maps all patterns of words in the given file onto
    # the set of words in the file having the pattern
    patterns = {}
    for word in open(filename, 'r'):

        word = word.strip()
        key = pattern(word)

        if len(word) >= length:

            if key in patterns:
                patterns[key].add(word)
            else:
                patterns[key] = {word}

    # filter dictionary using the given criteria
    return patterns if occurrences == 1 else {
        key:words
        for key, words in patterns.items()
        if len(words) >= occurrences
    }

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

9.4 Plutokiller

```
"""
>>> coordinates('photo1.txt')
{(10, 8), (5, 5), (6, 8), (6, 6), (7, 1), (10, 7), (9, 8), (10, 10), (6, 0), (1, 4), (0, 10),
    (1, 10), (5, 1), (8, 6), (10, 0), (9, 6), (2, 4), (7, 2), (8, 4)}
>>> coordinates('photo2.txt')
{(10, 8), (4, 7), (6, 8), (7, 1), (10, 7), (10, 10), (9, 8), (6, 0), (0, 7), (1, 4), (7, 7),
    (8, 7), (1, 10), (5, 1), (10, 0), (9, 6), (2, 4), (7, 2), (8, 4)}

>>> divergence('photo1.txt', 'photo2.txt')
({(8, 6), (5, 5), (0, 10), (6, 6)}, {(4, 7), (7, 7), (0, 7), (8, 7)})

>>> planets('photo1.txt', 'photo2.txt')
{(4, 7): {(5, 5), (6, 6)}, (8, 7): {(8, 6)}, (7, 7): {(8, 6), (6, 6)}, (0, 7): {(0, 10)}}

>>> print(comparator('photo1.txt', 'photo2.txt'))
-------n--o-
----*-----*-
----*-------
------------
-------n----
-*---o------
*-----o-*---
-**----n----
----*-on----
------*-*---
*------**-*-
"""

def coordinates(filename):

    # read image and create set containing coordinates of all stars
    coordinates = set()
    for row, line in enumerate(open(filename, 'r')):
        for col, star in enumerate(line):
            if star == '*':
```

```
                coordinates.add((row, col))

    # return set containing coordinates of all stars on the given image
    return coordinates

def divergence(old, new):

    # determine coordinates of all stars on the old image
    old = coordinates(old)

    # determine coordinates of all stars on the new image
    new = coordinates(new)

    # determine difference between old and new images
    return old - new, new - old

def planets(old, new):

    def distance(coord1, coord2):

        x1, y1 = coord1
        x2, y2 = coord2

        return (x1 - x2) ** 2 + (y1 - y2) ** 2

    # determine difference between old and new images
    old, new = divergence(old, new)

    # for each candidate planet on the new image, determine all nearest stars
    # on the old image
    candidates = {}
    for coord1 in new:
        nearest, stars = None, set()
        for coord2 in old:
            d = distance(coord1, coord2)
            if nearest is None or d < nearest:
                nearest = d
                stars = {coord2}
            elif d == nearest:
                stars.add(coord2)
        candidates[coord1] = stars

    return candidates

def comparator(old, new):

    # read old image from file
    starmap = [list(line.rstrip('\n')) for line in open(old, 'r')]

    # determine difference between old and new images (candidate planets)
    old, new = divergence(old, new)

    # mark old positions of planets using the letter o
    for x, y in old:
        starmap[x][y] = 'o'

    # mark old positions of planets using the letter n
    for x, y in new:
        starmap[x][y] = 'n'

    # return image with old and new positions of planets marked
    return '\n'.join(''.join(row) for row in starmap)

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

9.5 AWOL

```
def coordinates(filename):

    """
    >>> coords = coordinates('airports.csv')
    >>> len(coords)
    9187
    >>> type(coords)
    <class 'dict'>
    >>> coords['BRU']
    (50.902222, 4.485833)
    >>> coords['CDG']
    (49.016667, 2.55)
    >>> coords['DCA']
    (38.851944, -77.037778)
    >>> coords['LAX']
    (33.9425, -118.407222)
    """

    import csv

    # buid dictionary from given CSV file
    coords = {}
    handle = open(filename, 'r', encoding='utf-8')
    for row in csv.reader(handle, delimiter=','):
        coords[row[0]] = (float(row[5]), float(row[6]))

    # return dictionary that maps airports onto their coordinates
    return coords

def haversine(coord1, coord2):

    """
    >>> haversine((50.902222, 4.485833), (49.016667, 2.55)) # BRU <-> CDG
    251.2480027355068
    >>> haversine((38.851944, -77.037778), (33.9425, -118.407222)) # DCA <-> LAX
    3710.8262543589817
    """

    from math import sin, cos, radians, atan2, sqrt

    # define Earth radius
    r = 6371.0

    # unpack individual latitudes and lpngitudes
    b1, l1 = (radians(c) for c in coord1)
    b2, l2 = (radians(c) for c in coord2)

    # compute Haversine distance
    a = (
        sin((b2 - b1) / 2) ** 2 +
        cos(b1) * cos(b2) * sin((l2 - l1) / 2) ** 2
    )
    c = atan2(sqrt(a), sqrt(1 - a))

    return 2 * r * c

def flightplan(departure, arrival, coordinates, range=1000):

    """
    >>> coords = coordinates('airports.csv')

    >>> flightplan('DCA', 'LAX', coords)
    ['DCA', 'MTO', 'HLC', 'BFG', 'LAX']
    >>> flightplan('DCA', 'LAX', coords, range=2000)
    ['DCA', 'DDC', 'LAX']
    >>> flightplan('DCA', 'LAX', coords, 4000)
    ['DCA', 'LAX']
```

```
    >>> flightplan('BRU', 'CDG', coords)
    ['BRU', 'CDG']
    >>> flightplan('BRU', 'CDG', coords, range=50)
    Traceback (most recent call last):
    AssertionError: no possible route
    """

    plan = [departure]

    # extend flight plan until destination is reached
    while plan[-1] != arrival:

        # determine next airport
        airport, distance = None, None
        for code, coord in coordinates.items():
            if (
                code not in plan and
                haversine(coordinates[plan[-1]], coord) <= range and
                (distance is None or haversine(coord, coordinates[arrival]) < distance)
            ):
                airport = code
                distance = haversine(coord, coordinates[arrival])

        # check if next airport has been found
        assert distance is not None, 'no possible route'

        # append next airport to flight plan
        plan.append(airport)

    return plan

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

9.6 Sestina

```
def endword(line):

    """
    >>> endword("Lo ferm voler qu'el cor m'intra")
    'intra'
    >>> endword("no'm pot ges becs escoissendre ni ongla")
    'ongla'
    >>> endword("de lauzengier qui pert per mal dir s'arma;")
    'arma'
    """

    # find position after last word: step back to position before last letter
    stop = len(line)
    while not line[stop - 1].isalpha():
        stop -= 1

    # find position of first letter of last word: step back as long as there is
    # a letter before the current position, and stop as well if the start of the
    # line has been reached
    start = stop - 1
    while start > 0 and line[start - 1].isalpha():
        start -= 1

    # extract the last word from the line
    return line[start:stop]

def stanzas(filename):

    """
```

```
    >>> stanzas('sestina0.txt')
    [['intra', 'ongla', 'arma', 'verja', 'oncle', 'cambra'], ['cambra', 'intra', 'oncle', '
        ongla', 'verja', 'arma'], ['arma', 'cambra', 'verja', 'intra', 'ongla', 'oncle'], ['
        oncle', 'arma', 'ongla', 'cambra', 'intra', 'verja'], ['verja', 'oncle', 'intra', '
        arma', 'cambra', 'ongla'], ['ongla', 'verja', 'cambra', 'oncle', 'arma', 'intra'], ['
        oncle', 'arma', 'intra']]
    >>> stanzas('sestina1.txt')
    [['enters', 'nail', 'soul', 'rod', 'uncle', 'room'], ['room', 'enters', 'uncle', 'nail', '
        rod', 'soul'], ['soul', 'room', 'rod', 'enters', 'nail', 'uncle'], ['uncle', 'soul', '
        nail', 'room', 'enters', 'rod'], ['rod', 'uncle', 'enters', 'soul', 'room', 'nail'],
        ['nail', 'rod', 'room', 'uncle', 'soul', 'enters'], ['nail', 'soul', 'enters']]
    >>> stanzas('sestina2.txt')
    [['woe', 'sound', 'cryes', 'part', 'sleepe', 'augment'], ['augment', 'woe', 'sound', '
        cryes', 'part', 'sleepe'], ['sleepe', 'augment', 'woe', 'sound', 'cryes', 'part'], ['
        part', 'sleepe', 'augment', 'woe', 'sound', 'cryes'], ['cryes', 'part', 'sleepe', '
        augment', 'woe', 'sound'], ['sound', 'cryes', 'part', 'sleepe', 'augment', 'woe'], ['
        sound', 'part', 'augment']]
    """

    # initialize list of stanzas and list of endwords of the current stanza
    poem, stanza = [], []

    # process lines in the poem one by one
    for line in open(filename, 'r'):

        if line.strip():

            # determine endword, convert to lowercase and append to the list of
            # endwords of the current stanza
            stanza.append(endword(line).lower())

        else:

            # append list of endwords of the previous stanza to the poem
            if stanza:
                poem.append(stanza)

            # start a new list of endwords for the next stanza
            stanza = []

    # append list of endwords of the final stanza (if this wasn't done before)
    if stanza:
        poem.append(stanza)

    # return endwords of the stanzas of the poem
    return poem

def permutation(words, pattern=None):

    """
    >>> permutation(['rose', 'love', 'heart', 'sang', 'rhyme', 'woe'])
    ['woe', 'rose', 'rhyme', 'love', 'sang', 'heart']
    >>> permutation(['woe', 'rose', 'rhyme', 'love', 'sang', 'heart'])
    ['heart', 'woe', 'sang', 'rose', 'love', 'rhyme']
    >>> permutation(['rose', 'love', 'heart', 'sang', 'rhyme'])
    ['rhyme', 'rose', 'sang', 'love', 'heart']
    >>> permutation(['rose', 'love', 'heart', 'sang', 'rhyme', 'woe'], [6, 1, 5, 2, 4, 3])
    ['woe', 'rose', 'rhyme', 'love', 'sang', 'heart']
    >>> permutation(['rose', 'love', 'heart', 'sang', 'rhyme', 'woe'], [6, 5, 4, 3, 2, 1])
    ['woe', 'rhyme', 'sang', 'heart', 'love', 'rose']
    >>> permutation(['rose', 'love', 'heart', 'sang', 'rhyme', 'woe'], [6, 1, 5, 3, 4, 3])
    Traceback (most recent call last):
    AssertionError: invalid permutation
    """

    if pattern is None:

        # canonical representation if no pattern was given
        pattern = [
```

```
            (-1 if n % 2 else 1) * (n // 2)
            for n in range(1, len(words) + 1)
        ]

    else:

        # check if the given pattern is a permutation of the integers 1, 2, .. n
        # where n is the number of words in the given list of elements
        assert sorted(pattern) == list(range(1, len(words) + 1)), 'invalid permutation'

    return [words[i - 1] for i in pattern]

def sestina(filename, pattern=None):

    """
    >>> sestina('sestina0.txt')
    True
    >>> sestina('sestina0.txt', [6, 1, 5, 2, 4, 3])
    True
    >>> sestina('sestina1.txt')
    True
    >>> sestina('sestina2.txt')
    False
    >>> sestina('sestina2.txt', [6, 1, 2, 3, 4, 5])
    True
    """

    # determine stanzas and their endwords for the given poem
    endwords = stanzas(filename)
    if not endwords:
        return False

    # determine number of lines in the first stanza
    n = len(endwords[0])

    # check whether the number of stanzas equals n or n + 1
    if len(endwords) not in {n, n + 1}:
        return False

    # check whether the endwords of each stanza result from applying the given
    # permutation to the endwords of the previous stanza
    # NOTE: this also implicitly checks that each stanza has n lines
    for stanza in range(1, n):
        if endwords[stanza] != permutation(endwords[stanza - 1], pattern):
            return False

    # check if the envoi (if present) complies to the rules
    if len(endwords) == n + 1:

        # check if the envoi has n // 2 lines
        if len(endwords[-1]) != n // 2:
            return False

        # check if endwords of the envoi are also endwords of the other stanzas
        if not set(endwords[-1]) <= set(endwords[0]):
            return False

    # all rules for sestina-like poems have been fulfilled
    return True

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

10 Series 10: object-oriented programming

10.1 ISBN

```
class ISBN13:

    """
    >>> code = ISBN13(9780136110675)
    >>> print(code)
    978-0-13611067-5
    >>> code
    ISBN13(9780136110675, 1)
    >>> code.isValid()
    True
    >>> code.asISBN10()
    '0-13611067-3'
    """

    def __init__(self, code, length=1):

        # check validity of arguments
        assert isinstance(code, int), 'ISBN-13 codes must only contain digits.'
        assert len(str(code)) == 13, 'ISBN13-codes must contain 13 digits.'
        assert 1 <= length <= 5, 'The specification of the country group of an ISBN-13 code
            must have 1 to 5 digits.'

        # object properties: ISBN-code and length of country group
        self.code = str(code)   # convert to string
        self.length = length

    def __str__(self):

        # return formatted representation of ISBN-code
        return '{}-{}-{}-{}'.format(
            self.code[:3],
            self.code[3:3 + self.length],
            self.code[3 + self.length:-1],
            self.code[-1]
        )

    def __repr__(self):

        # return string containing a Python expression that results in a new
        # object having the same internal state as the current object
        return 'ISBN13({}, {})'.format(self.code, self.length)

    def isValid(self):

        def checkdigit(code):

            # compute ISBN-13 check digit
            check = sum((3 if i % 2 else 1) * int(code[i]) for i in range(12))

            # convert the check digit into string representation
            return str((10 - check) % 10)

        # check validity of check digit
        return self.code[12] == checkdigit(self.code)

    def asISBN10(self):

        def checkdigit(code):

            # compute ISBN-10 check digit
            check = sum((i + 1) * int(code[i]) for i in range(9)) % 11

            # convert the check digit into string representation
            return 'X' if check == 10 else str(check)
```

```
        # return no result for invalid ISBN-13 codes
        if not self.isValid() or str(self.code)[:3] == '979':
            return None

        # convert ISBN-13 code into ISBN-10 code
        code = self.code[3:-1]
        check = checkdigit(code)
        return '{}-{}-{}'.format(
            code[:self.length],
            code[self.length:],
            check
        )

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

10.2 Scrabble's secret message

```
class Bag:

    """
    >>> bag = Bag('
        IAMDIETINGIEATQUINCEJELLYLOTSOFGROUNDMAIZEGIVESVARIETYICOOKRHUBARBANDSODAWEEPANEWORPUTONEXTRAFLESH_
        ')
    >>> bag.content == {'U': 4, '_': 2, 'C': 2, 'K': 1, 'D': 4, 'T': 6, 'Q': 1, 'V': 2, 'A':
        9, 'F': 2, 'O': 8, 'J': 1, 'I': 9, 'N': 6, 'P': 2, 'S': 4, 'M': 2, 'W': 2, 'E': 12, 'Z
        ': 1, 'G': 3, 'Y': 2, 'B': 2, 'L': 4, 'R': 6, 'X': 1, 'H': 2}
    True
    >>> print(bag)
    1: JKQXZ
    2: BCFHMPVWY_
    3: G
    4: DLSU
    6: NRT
    8: O
    9: AI
    12: E
    >>> bag
    Bag('
        AAAAAAAAABBCCDDDDEEEEEEEEEEEEFFGGGHHIIIIIIIIIJKLLLLMMNNNNNNOOOOOOOOPPQRRRRRRSSSSTTTTTTUUUUVVWWXYYZ_
        ')
    >>> bag.remove('AEERTYOXMCNB_S')
    >>> print(bag)
    1: BCJKMQYZ_
    2: FHPVW
    3: GS
    4: DLU
    5: NRT
    7: O
    8: A
    9: I
    10: E
    >>> bag
    Bag('
        AAAAAAAABCDDDDEEEEEEEEEEFFGGGHHIIIIIIIIIJKLLLLMNNNNNOOOOOOOPPQRRRRRSSSTTTTTUUUUVVWWYZ_
        ')
    >>> bag.remove('XXX')
    Traceback (most recent call last):
    AssertionError: not all letters are in the bag
    >>> print(bag)
    1: BCJKMQYZ_
    2: FHPVW
    3: GS
    4: DLU
    5: NRT
```

```
    7: O
    8: A
    9: I
    10: E
    >>> bag
    Bag('
        AAAAAAAABCDDDDEEEEEEEEEEFFGGGHHIIIIIIIIIJKLLLLMNNNNNOOOOOOOPPQRRRRRSSSTTTTTUUUUVVWWYZ_
        ')
    """

    def __init__(self, letters=''):

    # bag is represented as frequency table built from the given letters
        self.content = {}
        for letter in letters:
            self.content[letter] = self.content.get(letter, 0) + 1

    def __str__(self):

        # reverse dictionary that represents bag, with keys grouped into set
        bag, overzicht = self.content, {}
        for letter, count in bag.items():
            if count in overzicht:
                overzicht[count].add(letter)
            else:
                overzicht[count] = {letter}

        # output overview of letters in the bag, grouped by number of occurrences
        return '\n'.join(
            '{}: {}'.format(letter, ''.join(sorted(overzicht[letter])))
            for letter in sorted(overzicht)
        )

    def __repr__(self):

        # generate string representation with letters sorted alphabetically
        return 'Bag({!r})'.format(''.join(
            letter * count for letter, count in sorted(self.content.items())
        ))

    def remove(self, letters):

         # check if all given letters are in the bag
        bag, nodig = self.content, Bag(letters).content
        assert all(
            letter in bag and nodig[letter] <= bag[letter]
            for letter in nodig
        ), 'not all letters are in the bag'

        # remove given letters from the bag
        for letter in letters:
            if bag[letter] == 1:
                del bag[letter]
            else:
                bag[letter] -= 1

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

10.3 Geheimschreiber

```
class T52:

    """
    >>> machine1 = T52(3, 5, 'ABCDEFGHIJKLMNOPQRSTUVWXYZ')
```

```
    >>> machine1.encodeSymbol('G')
    'X'
    >>> machine1.encodeSymbol('S')
    'H'
    >>> machine1.encodeSymbol('-')
    '_'

    >>> machine1.decodeSymbol('X')
    'G'
    >>> machine1.decodeSymbol('H')
    'S'
    >>> machine1.decodeSymbol('-')
    '_'

    >>> machine1.encode('G-SCHREIBER')
    'X-HLAERDIRE'
    >>> machine1.decode('X-HLAERDIRE')
    'G-SCHREIBER'

    >>> machine2 = T52(17, 11, 'ABCDEFGHIJKLMNOPQRSTUVWXYZ')
    >>> machine2.encode('X-HLAERDIRE')
    'M-AQLBOKROB'

    >>> machine12 = machine1 + machine2
    >>> machine12.encode('G-SCHREIBER')
    'M-AQLBOKROB'

    >>> T52(4, 5, 'ABCDEFGHIJKLMMLKJIHGFEDCBA')
    Traceback (most recent call last):
    AssertionError: alphabet has repeated symbols

    >>> T52(4, 5, 'ABCDEFGHIJKLMNOPQRSTUVWXYZ')
    Traceback (most recent call last):
    AssertionError: 4 and 26 are not coprime

    >>> machine1 + T52(17, 11, 'abcdefghijklmnopqrstuvwxyz')
    Traceback (most recent call last):
    AssertionError: alphabets are different
    """

    def __init__(self, a, b, alphabet):

        # store encryption keys as object properties
        self.a = a
        self.b = b

        # check if all symbols of the alphabet are different and store the
        # alphabet as an object property
        m = len(alphabet)
        assert m == len(set(alphabet)), 'alphabet has repeated symbols'
        self.alphabet = alphabet

        # check if a and m are coprime
        from math import gcd
        assert gcd(a, m) == 1, '{} and {} are not coprime'.format(a, m)

        # construct dictionary for encoding of symbols in the alphabet
        self.enc = {
            s: alphabet[(a * x + b) % m] for x, s in enumerate(alphabet)
        }

        # construct dictionary for decoding of symbols in the alphabet
        # NOTE: inverse dictionary of the one used for encoding
        self.dec = {c: o for o, c in self.enc.items()}

    def encodeSymbol(self, symbol):

        return self.enc.get(symbol, symbol)
```

```
    def decodeSymbol(self, symbol):

        return self.dec.get(symbol, symbol)

    def encode(self, message):

        return ''.join(self.encodeSymbol(symbol) for symbol in message)

    def decode(self, message):

        return ''.join(self.decodeSymbol(symbol) for symbol in message)

    def __add__(self, other):

        assert self.alphabet == other.alphabet, 'alphabets are different'

        a1, b1 = self.a, self.b
        a2, b2 = other.a, other.b

        return T52(a1 * a2, a2 * b1 + b2, self.alphabet)

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

10.4 Racetrack Playa

```
class Block:

    """
    >>> rock = Block(5, 2, 3)
    >>> rock
    Block(length=5, height=2, width=3, position=(0, 0))
    >>> rock.area()
    62.0
    >>> rock.volume()
    30.0
    >>> rock.diagonal()
    6.164414002968976
    >>> rock2 = rock.slide('R')
    >>> rock2
    Block(length=5, height=2, width=3, position=(0, 5))
    >>> rock is rock2
    True
    >>> rock.slide('F')
    Block(length=5, height=2, width=3, position=(3, 5))
    >>> rock.tilt('L')
    Block(length=2, height=5, width=3, position=(3, 3))
    >>> rock.tilt('B')
    Block(length=2, height=3, width=5, position=(0, 3))
    >>> rock.tilt('B').slide('L').tilt('L').slide('B')
    Block(length=5, height=2, width=3, position=(-8, -4))
    >>> rock.sail('SB')
    Block(length=5, height=2, width=3, position=(-11, -4))
    >>> rock.sail('TR')
    Block(length=2, height=5, width=3, position=(-11, 1))
    >>> rock.sail('SFSFTLSLTBTBSRSFTRTFTRTRSBSF')
    Block(length=2, height=3, width=5, position=(-2, 6))

    >>> rock.tilt('X')
    Traceback (most recent call last):
    AssertionError: invalid direction
    >>> rock.sail('XY')
    Traceback (most recent call last):
    AssertionError: invalid movement
    >>> rock.sail('TY')
    Traceback (most recent call last):
```

```
    AssertionError: invalid direction
    """

    def __init__(self, length, height, width, position=(0, 0)):

        # object properties: length, height and width
        self.L, self.H, self.B = L, H, B = length, height, width

        # object property: position of block
        self.x, self.y = tuple(position)

        # precompute area, volume and diagonal (never change)
        self._area = 2.0 * (L * B + L * H + H *B)
        self._volume = float(L * H * B)
        self._diagonal = (L ** 2 + H ** 2 + B ** 2) ** 0.5

    def area(self):

        return self._area

    def volume(self):

        return self._volume

    def diagonal(self):

        return self._diagonal

    def __repr__(self):

        return '{}(length={}, height={}, width={}, position={})'.format(
            self.__class__.__name__,
            self.L,
            self.H,
            self.B,
            (self.x, self.y)
        )

    def slide(self, direction):

        # slide block in given direction
        if direction == 'R':
            self.y += self.L
        elif direction == 'L':
            self.y -= self.L
        elif direction == 'F':
            self.x += self.B
        elif direction == 'B':
            self.x -= self.B
        else:
            raise AssertionError('invalid direction')

        # return object reference
        return self

    def tilt(self, direction):

        # tilt block in given direction
        if direction in 'LR':
            self.y += self.L if direction == 'R' else -self.H
            self.L, self.H = self.H, self.L
        elif direction in 'FB':
            self.x += self.H if direction == 'F' else -self.B
            self.B, self.H = self.H, self.B
        else:
            raise AssertionError('invalid direction')

        # return object reference
        return self
```

输出class name

可以从method中实例化

交换数值

```
    def sail(self, directions):

        # move block in given directions
        for i in range(0, len(directions), 2):
            movement, direction = directions[i:i + 2]
            if movement == 'S':
                self.slide(direction)
            elif movement == 'T':
                self.tilt(direction)
            else:
                raise AssertionError('invalid movement')

        # return object reference
        return self

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

10.5 Data compression

```
class ZIP:

    """
    >>> zip = ZIP('codes.txt')

    >>> zip.symbol2bitstring('i')
    '1000'
    >>> zip.symbol2bitstring('e')
    '000'
    >>> zip.symbol2bitstring('T')
    Traceback (most recent call last):
    AssertionError: unknown symbol "T"

    >>> zip.bitstring2symbol('1000')
    'i'
    >>> zip.bitstring2symbol('000')
    'e'
    >>> zip.bitstring2symbol('01')
    Traceback (most recent call last):
    AssertionError: invalid bitstring

    >>> zip.compress('internet')
    '10000010011000011000001000000110'
    >>> len(zip.compress('internet'))
    31
    >>> zip.compress('internet explorer')
    '1000001001100001100000100000110111000100101001111001001101100000011000'
    >>> zip.compress('mozilla firefox')
    Traceback (most recent call last):
    AssertionError: unknown symbol "z"

    >>> zip.decompress('10000010011000011000001000000110')
    'internet'
    >>> zip.decompress
        ('1000001001100001100000100000110111000100101001111001001101100000011000')
    'internet explorer'
    >>> zip.decompress('10000010011000011000001000001101')
    Traceback (most recent call last):
    AssertionError: invalid bitstring
    >>> zip.decompress('10000010011000011000000000110')
    Traceback (most recent call last):
    AssertionError: invalid bitstring
    """

    def __init__(self, filename):
```

```
        # dictionary that maps symbols onto their corresponding bitstring
        self._symbol2bitstring = {}

        # dictionary that maps bitstrings onto their corresponding symbol
        self._bitstring2symbol = {}

        # construct dictionaries from file contents
        for regel in open(filename, 'r'):
            # note: because the symbol might be a tab itself, it would be
            #       unsafe to split the line with tabs as a delimiter
            symbol, bitstring = regel[0], regel[2:].rstrip('\n')
            self._symbol2bitstring[symbol] = bitstring
            self._bitstring2symbol[bitstring] = symbol

    def symbol2bitstring(self, symbol):

        try:
            return self._symbol2bitstring[symbol]
        except KeyError as e:
            raise AssertionError('unknown symbol "{}"'.format(e.args[0]))

    def bitstring2symbol(self, bitstring):

        try:
            return self._bitstring2symbol[bitstring]
        except:
            raise AssertionError('invalid bitstring')

    def compress(self, text):

        return ''.join(self.symbol2bitstring(symbol) for symbol in text)

    def decompress(self, bitstring):

        text = ''
        while bitstring:    没找完就不停止

            # find prefix that corresponds to a symbol      寻数类问题:
            prefix = 1  每找到一个之后就重置prefix          需要不断增长数值, 不断在hash table中寻找
            while (
                prefix <= len(bitstring) and
                bitstring[:prefix] not in self._bitstring2symbol
            ):                  由于不断的阶段, 所以总是可以从头开始
                prefix += 1

            # add symbol corresponding to prefix (if found)
            if prefix <= len(bitstring):
                bits = bitstring[:prefix]
                text += self.bitstring2symbol(bits)
                bitstring = bitstring[len(bits):]   随着不断的找到, 未知seq 在不断的缩小
            else:
                raise AssertionError('invalid bitstring')

        return text

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

10.6 Lemmings

```
class Lemming:

    """
    >>> lemming = Lemming('level.txt', 3, '<')
    >>> lemming.position()
```

```
(3, 3, '<')
>>> print(lemming)
##################################
#                                #
#         #                      #
#  <    ###                      #
###########      ###             #
###########    ########          #
###########  ##############      #
##################################
>>> lemming.step()
(3, 2, '<')
>>> lemming.step()
(3, 1, '<')
>>> lemming.step()
(3, 1, '>')
>>> lemming.step()
(3, 2, '>')
>>> print(lemming)
##################################
#                                #
#         #                      #
# >     ###                      #
###########      ###             #
###########    ########          #
###########  ##############      #
##################################
>>> lemming.steps(5)
[(3, 3, '>'), (3, 4, '>'), (3, 5, '>'), (3, 6, '>'), (3, 7, '>')]
>>> print(lemming)
##################################
#                                #
#         #                      #
#      >###                      #
###########      ###             #
###########    ########          #
###########  ##############      #
##################################
>>> lemming.step()
(2, 8, '>')
>>> print(lemming)
##################################
#                                #
#       > #                      #
#       ###                      #
###########      ###             #
###########    ########          #
###########  ##############      #
##################################
>>> lemming.step()
(2, 9, '>')
>>> lemming.step()
(1, 10, '>')
>>> print(lemming)
##################################
#         >                      #
#         #                      #
#       ###                      #
###########      ###             #
###########    ########          #
###########  ##############      #
##################################
>>> lemming.step()
(6, 11, '>')
>>> print(lemming)
##################################
#                                #
#         #                      #
#       ###                      #
```

```
    ###########      ###            #
    ###########    ########         #
    ###########> ##############     #
    ################################
    >>> lemming.steps(21)
    [(6, 12, '>'), (5, 13, '>'), (5, 14, '>'), (4, 15, '>'), (4, 16, '>'), (3, 17, '>'), (3,
        18, '>'), (3, 19, '>'), (4, 20, '>'), (4, 21, '>'), (4, 22, '>'), (5, 23, '>'), (5,
        24, '>'), (5, 25, '>'), (5, 26, '>'), (6, 27, '>'), (6, 28, '>'), (6, 29, '>'), (6,
        30, '>'), (6, 31, '>'), (6, 31, '<')]
    >>> lemming.steps(21)
    [(6, 30, '<'), (6, 29, '<'), (6, 28, '<'), (6, 27, '<'), (5, 26, '<'), (5, 25, '<'), (5,
        24, '<'), (5, 23, '<'), (4, 22, '<'), (4, 21, '<'), (4, 20, '<'), (3, 19, '<'), (3,
        18, '<'), (3, 17, '<'), (4, 16, '<'), (4, 15, '<'), (5, 14, '<'), (5, 13, '<'), (6,
        12, '<'), (6, 11, '<'), (6, 11, '>')]
    >>> print(lemming)
    ################################
    #                              #
    #         #                    #
    #       ###                    #
    ###########      ###            #
    ###########    ########         #
    ###########> ##############     #
    ################################
    """

    def __init__(self, level, column, direction):

        # read level description from file
        self._level = [regel.rstrip('\n') for regel in open(level, 'r')]

        # check validity of the level
        assert len(self._level) > 2 and len(self._level[0]) > 2, 'invalid level'

        # check validity of the starting position
        assert 0 < column < len(self._level[0]) - 1, 'invalid postion'

        # check validity of the initial direction
        assert direction in '<>', 'invalid direction'

        # setup initial position and direction
        self._row, self._col = 0, column
        self._direction = 1 if direction == '>' else -1

        # drop lemming until it stands on solid ground
        self.drop()

    def direction(self):

        return '>' if self._direction == 1 else '<'

    def position(self):

        return self._row, self._col, self.direction()

    def __str__(self):

        return '\n'.join(
            self._level[r] if r != self._row else self._level[r][:self._col] + self.direction
                () + self._level[r][self._col + 1:]
            for r in range(len(self._level))
        )

    def drop(self):

        # drop lemming until it stands on solid ground
        while self._level[self._row + 1][self._col] != '#':
            self._row += 1

    def step(self):
```

1, 同行, 前方如果是空, 就前进, drop(), 因为, 无论是掉还是不掉, 都这么做, 返回的结果都正确

```
        if self._level[self._row][self._col + self._direction] == ' ':
            self._col += self._direction     # move forward
            self.drop()                      # drop until on solid ground
        elif self._level[self._row - 1][self._col + self._direction] == ' ':
            self._col += self._direction     # move forward
            self._row -= 1                   # move up
        else:
            self._direction *= -1            # turn around

        return self.position()

    def steps(self, aantal):

        return [self.step() for _ in range(aantal)]

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

2, 注意elif, 如果前方不是空的, 那就检测前方往上是不是空的, 如果是空的,还能走

3, 否则就是反向

direction的目的是向左的同时, 不用把向左向右分开, direction, 向右就是+1, 向左就是 -1, +direction就可以把向左向右都解决了, 但是要另外设置一个method, 把direction和+1 -1随时切换