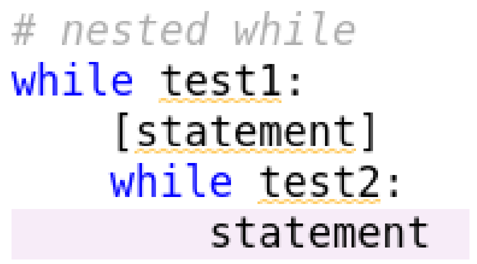
Nested loops

- we just saw example of nested loops

The syntax is:



```
# nested while
while test1:
    [statement]
    while test2:
        statement
```

The key point:

inner loop does **all** of its iterations for **each** of the outer loop iterations!!!

Python's other repetition structure: for loops

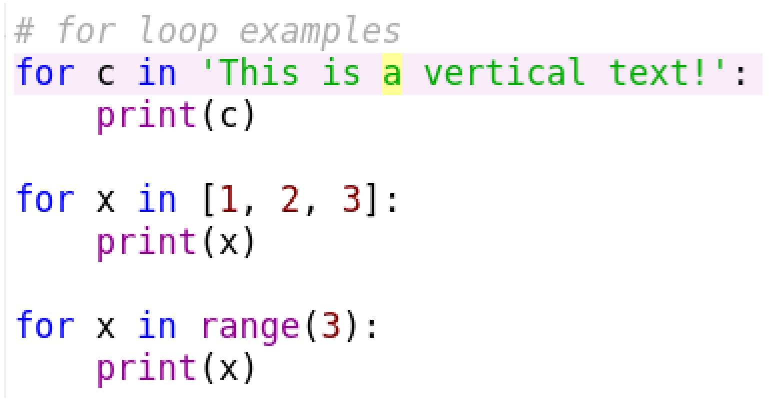
The syntax is:

```
for item in sequence:
    statement
```

"for each item in sequence execute the following statement[s]"

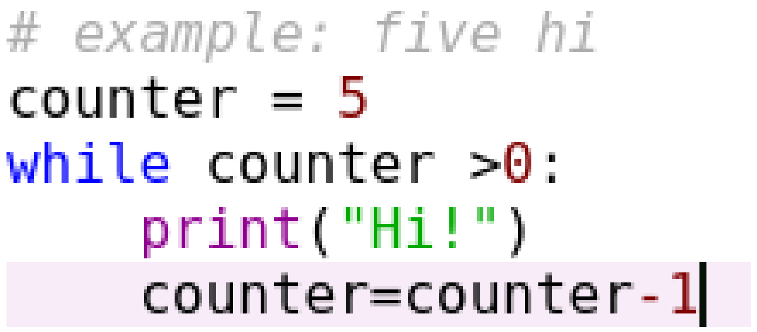
Examples of sequences (iterables):

- string	’hello’
- list	[1, 2, 3]
- tuple	(1, 2, 3)
- dictionary	{‘a’: 97, ‘b’: 98}
- set	{"apple", "banana", "cherry"}

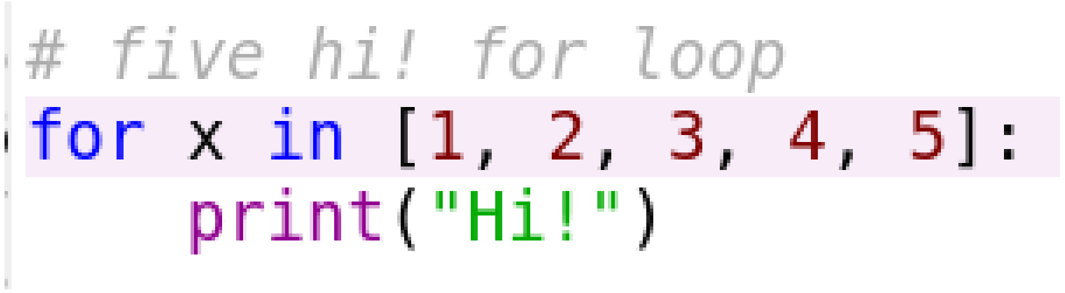


```
# for loop examples
for c in 'This is a vertical text!':
    print(c)

for x in [1, 2, 3]:
    print(x)

for x in range(3):
    print(x)
```

Five hi! revisited



```
# example: five hi
counter = 5
while counter >0:
    print("Hi!")
    counter=counter-1
```



```
# five hi! for loop
for x in [1, 2, 3, 4, 5]:
    print("Hi!")
```

- Not all repetitions can be represented using **for** loop
- **while** is more general and flexible,

Nested for loops
Exercise: draw a square

Write a program that reads in the size of a square and a character and then prints out a hollow square of that size out of the specified character and blanks. Your program should work for squares of all sizes from 1 to 20 inclusive. Here is a sample run of the program,

```
What size square? 5
What character? *
*****
*   *
*   *
*   *
*****
```

Think about pseudocode first!

Hint: **print(c, end='')** prints a character without moving on to the next line
(note default end='\n' &
print(c) => used default end character.

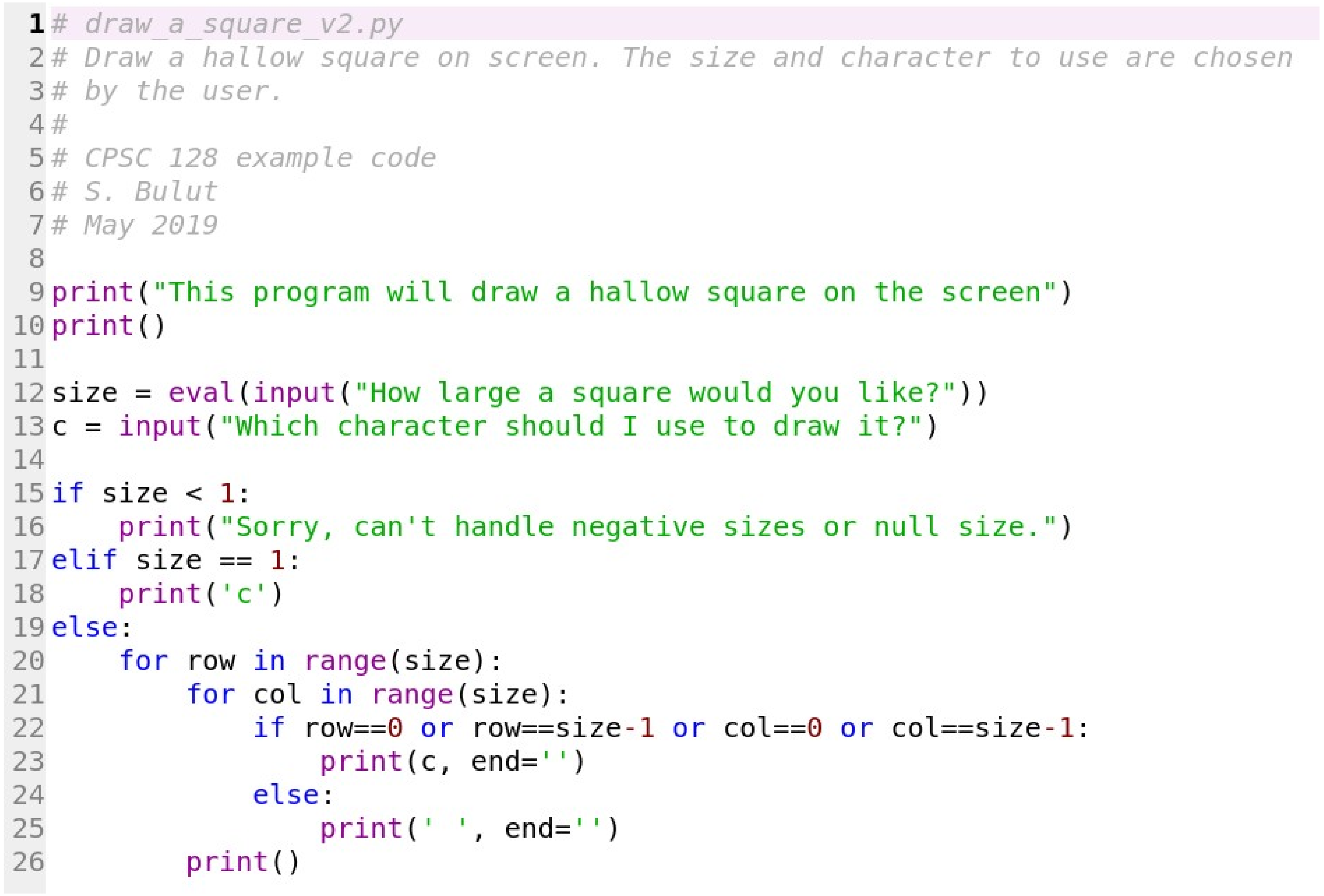
Hint: you might want to use **range()** to generate a sequence.

Nested for loops
Exercise: draw a square (solution v1)

draw_a_square_v1.py

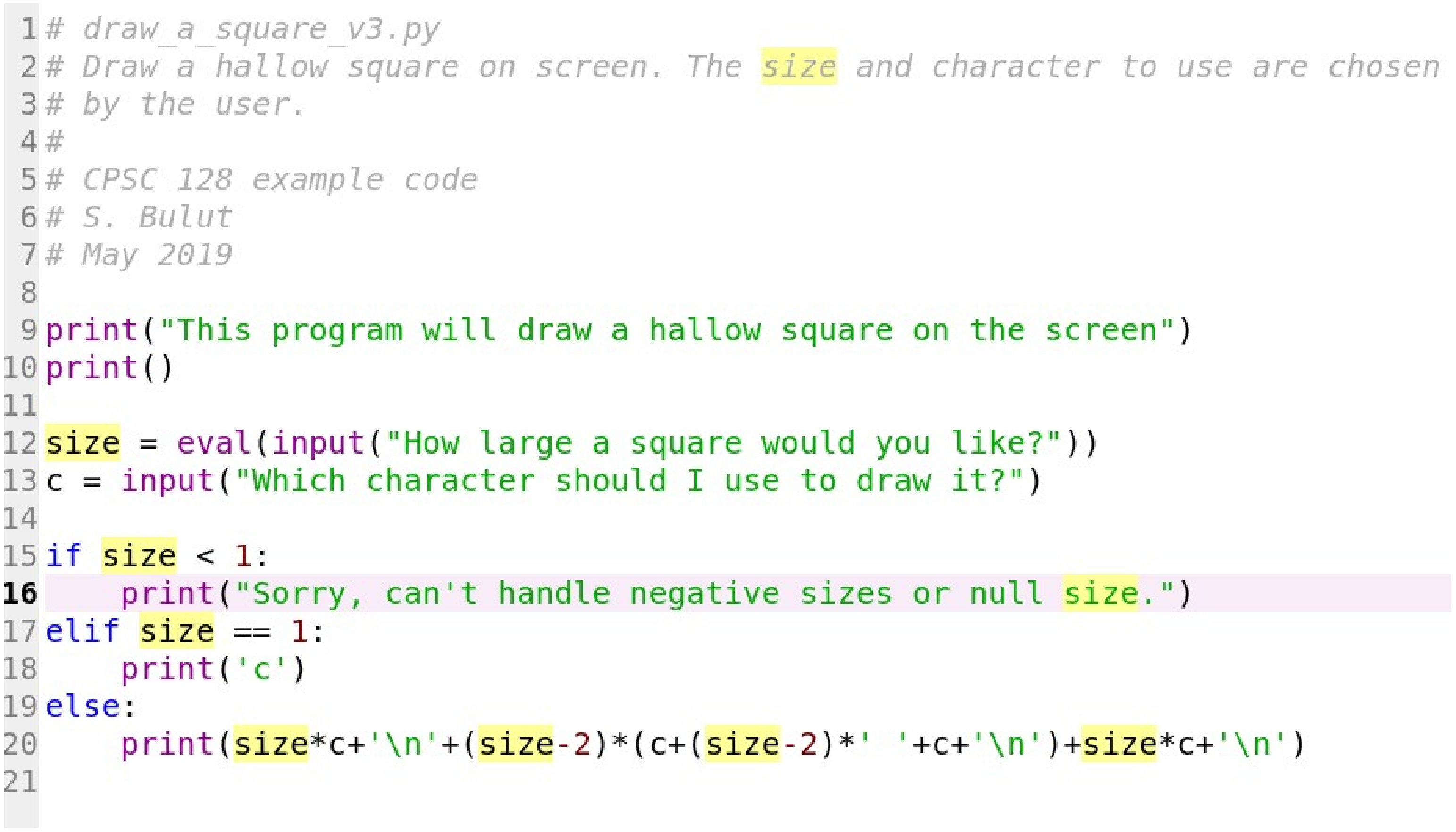
```python
print("This program will draw a hallow square on the screen")
print()

size = eval(input("How large a square would you like?"))
c = input("Which character should I use to draw it?")

if size < 1:
    print("Sorry, can't handle negative sizes or null size.")
elif size == 1:
    print('c')
else:
    # draw top row of the square
    for i in range(size):
        print(c, end='')
    print()

#    # draw the middle rows of the square
    for i in range(size-2):
        print(c, end='')
        for j in range(size-2):
            print(' ',end='')
        print(c)

    # draw the bottom row of the square
    for i in range(size):
        print(c, end='')
    print
```

Nested for loops
Exercise: draw a square (solution v2)



```python
# draw_a_square_v2.py
# Draw a hallow square on screen. The size and character to use are chosen
# by the user.
#
# CPSC 128 example code
# S. Bulut
# May 2019

print("This program will draw a hallow square on the screen")
print()

size = eval(input("How large a square would you like?"))
c = input("Which character should I use to draw it?")

if size < 1:
    print("Sorry, can't handle negative sizes or null size.")
elif size == 1:
    print('c')
else:
    for row in range(size):
        for col in range(size):
            if row==0 or row==size-1 or col==0 or col==size-1:
                print(c, end='')
            else:
                print(' ', end='')
        print()
```

Nested for loops
Exercise: draw a square (solution v3)



```python
# draw_a_square_v3.py
# Draw a hallow square on screen. The size and character to use are chosen
# by the user.
#
# CPSC 128 example code
# S. Bulut
# May 2019

print("This program will draw a hallow square on the screen")
print()

size = eval(input("How large a square would you like?"))
c = input("Which character should I use to draw it?")

if size < 1:
    print("Sorry, can't handle negative sizes or null size.")
elif size == 1:
    print('c')
else:
    print(size*c+'\n'+(size-2)*(c+(size-2)*' '+c+'\n')+size*c+'\n')

```

Summary: repetion structures in Python

```
while test-expression:
    statement
for item in sequence:
    statement
```

Applications of loops: Brute force method

When in doubt, use brute force. ~ Ken Thompson, Bell Labs

- We are not quick in doing processing and we are unreliable in doing repetitive operations.
- However, computers are very quick and reliable in doing repetitive calcs

- Hence, using loops, we can find brute force solutions for many computational problems

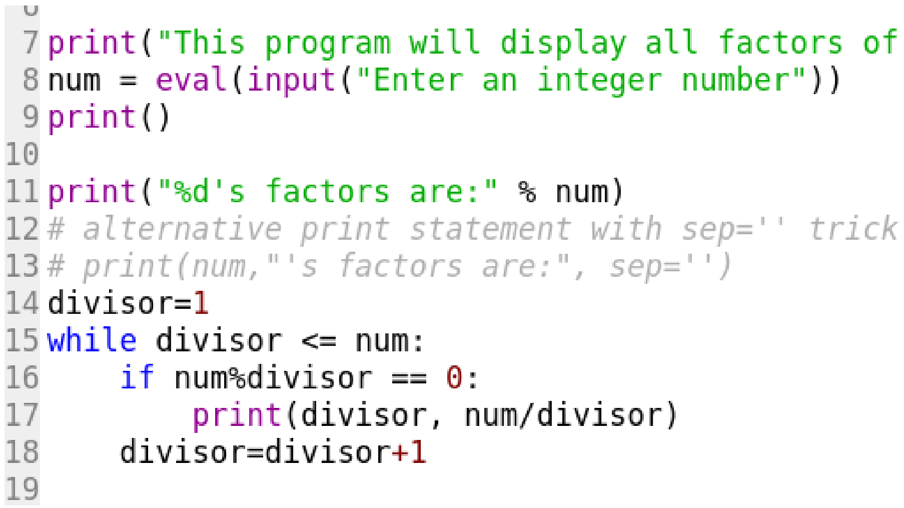
- Let’s consider examples of simulations and number crunching

Number crunching: Programming exercise: Factors

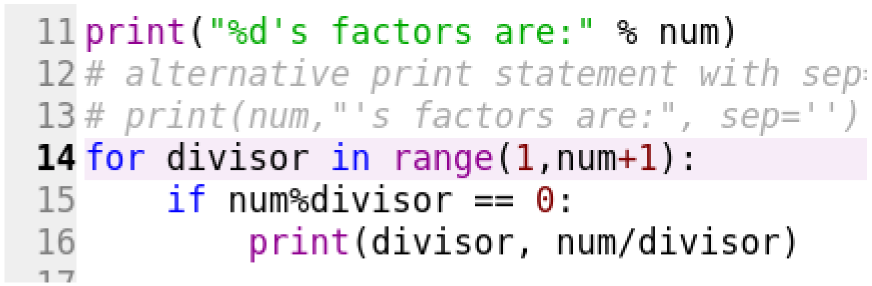
Write a program that inputs a positive whole number, and displays all the number's factors, i.e. all the numbers that divide into it exactly. For instance if the number 12 is input, the values 1, 2, 3, 4, 6, and 12 should be output.

Number crunching:
Programming exercise: Factors

Write a program that inputs a positive whole number, and displays all the number's factors, i.e. all the numbers that divide into it exactly. For instance if the number 12 is input, the values 1, 2, 3, 4, 6, and 12 should be output.



```
print("This program will display all factors of
num = eval(input("Enter an integer number"))
print()

print("%d's factors are:" % num)
# alternative print statement with sep='' trick
# print(num,"'s factors are:", sep='')
divisor=1
while divisor <= num:
    if num%divisor == 0:
        print(divisor, num/divisor)
    divisor=divisor+1

```



```
print("%d's factors are:" % num)
# alternative print statement with sep
# print(num,"'s factors are:", sep='')
for divisor in range(1,num+1):
    if num%divisor == 0:
        print(divisor, num/divisor)
```

Simulation

In a simulation problem we write a program that simulates a real world process, relying on our ability to generate random numbers to take account of the uncertainty of real-world processes like whether

- newborns are girls or boys,
- how long it will take to unload an airplane,
- the chances a vehicle will turn left or right or go straight.

If we do this enough times we can make a reliable estimate of the probability of a real-world event occurring

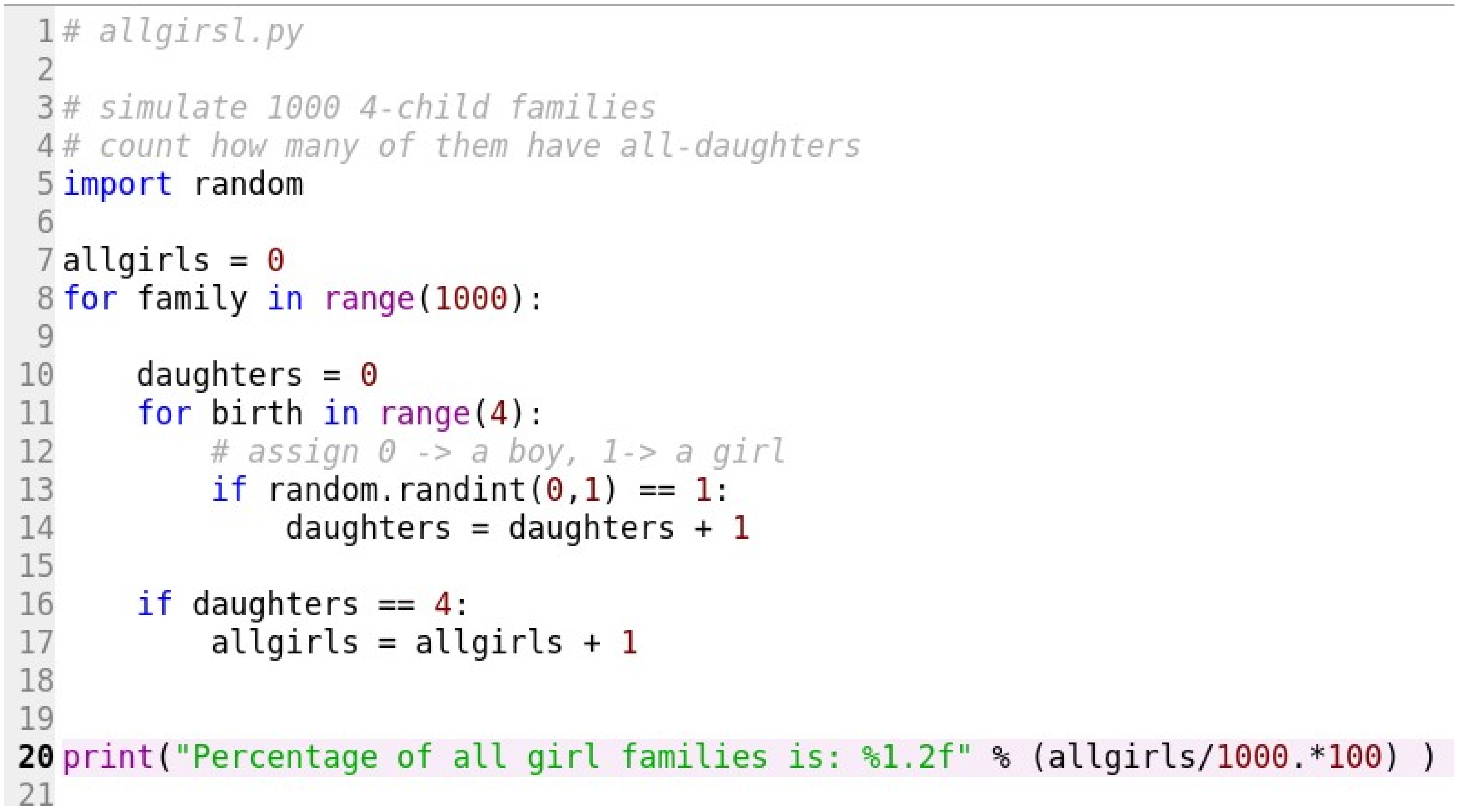
Problem: What are the chances: four children, all girls?

Write a simulation program that estimates the percentage of 4-child families in which all the children are girls by simulating the formation of 1000 four-child families. Assume that the chances of a newborn being a girl or a boy are equal. A sample run of the program might look like this:

```
In 1,000 simulated four-child families
approximately 7% were made up of four daughters.
```

Hint: use ‘random.randint()’ to flip a coin to see if
a birth event will yield a boy or a girl

Problem: What are the chances: four children, all girls?



```python
# allgirsl.py

# simulate 1000 4-child families
# count how many of them have all-daughters
import random

allgirls = 0
for family in range(1000):

    daughters = 0
    for birth in range(4):
        # assign 0 -> a boy, 1-> a girl
        if random.randint(0,1) == 1:
            daughters = daughters + 1

    if daughters == 4:
        allgirls = allgirls + 1


print("Percentage of all girl families is: %1.2f" % (allgirls/1000.*100) )

```

Re-do: using while loop!

Part II Object based programming

- Aggregate data types 1: Lists and strings.
- Functions.
- Aggregate data types 2: Dictionaries.
- Text files.

Object based programming

- all the values that we've worked with are obj's

- We did things like 'x = a + 10' using built-in operators,

- We did polymorhism/operator overloading

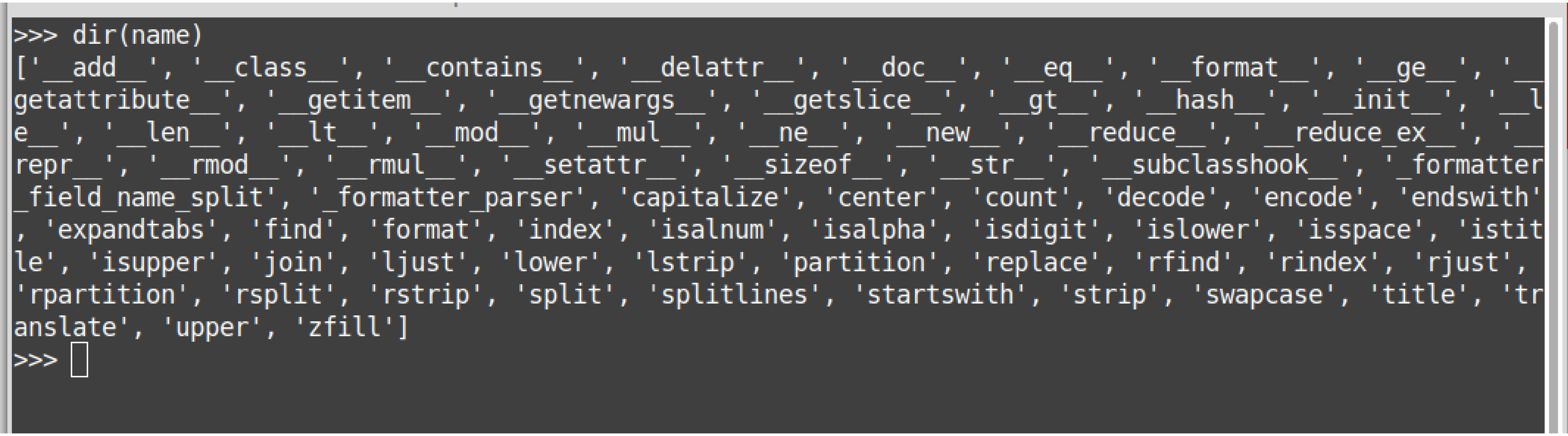
```
x = a + 10        vs  s='hello' + 'world'
9%5               vs  print("%d apples" % 9)
```

In python everything is an object:

An object is a collection of attributes and methods/functions that it knows how to apply to itself

Object based programming

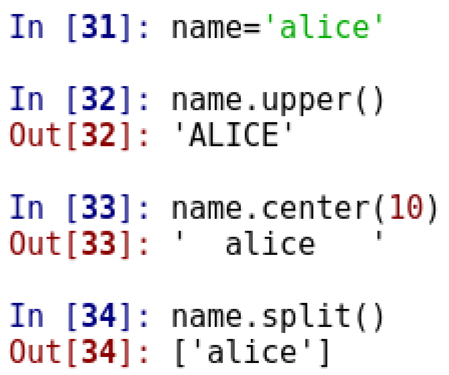
Example: name=’alice’ is a string object
its attribute is a being sequence of the characters ‘a’ ‘l’ ‘i’ ‘c’ ‘e’
its methods:



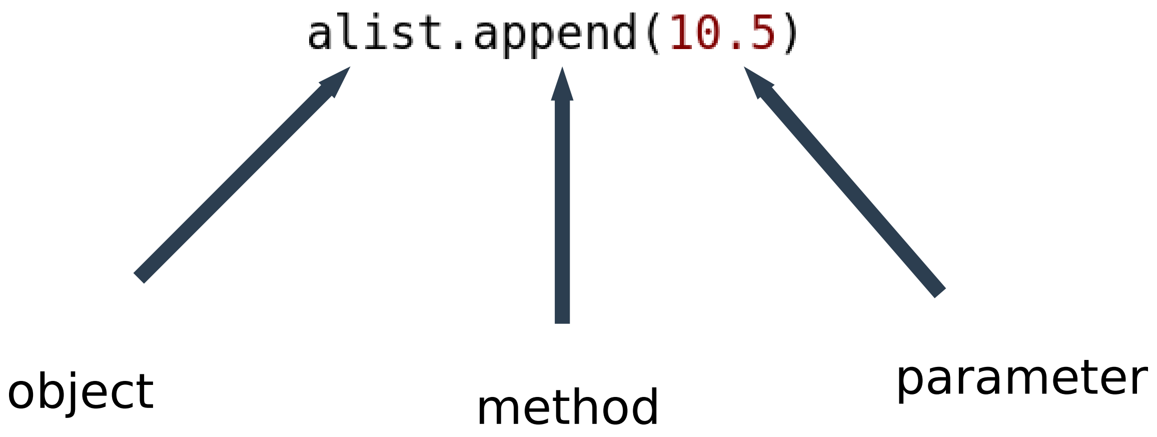
```
>>> dir(name)
['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__', '__format__', '__ge__', '__
getattribute__', '__getitem__', '__getnewargs__', '__getslice__', '__gt__', '__hash__', '__init__', '__l
e__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__
repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '_formatter
_field_name_split', '_formatter_parser', 'capitalize', 'center', 'count', 'decode', 'encode', 'endswith'
, 'expandtabs', 'find', 'format', 'index', 'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace', 'istit
le', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'partition', 'replace', 'rfind', 'rindex', 'rjust',
'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'tr
anslate', 'upper', 'zfill']
>>> 
```

- methods with double underscore ‘__’ correspond to methods invoked by an operat
such as ‘+’ corresponding to ‘__add__’
- others invoked via syntax: `object_name.method_name(parameters)`

Object based programming



```
In [31]: name='alice'

In [32]: name.upper()
Out[32]: 'ALICE'

In [33]: name.center(10)
Out[33]: '  alice   '

In [34]: name.split()
Out[34]: ['alice']
```



```
alist.append(10.5)
```

Sequence Types

Do the following examples:

Strings

```
name = "Joe"
"m" in name
"o" not in name
min(name)
max(name)
name = name + " " + "Sixpack"
print(name)
name[0]
name[2]
name[-1]
name[-2]
name[2:7]
len(name)
name*3
```

Lists

```
nums = [1, 2, 3, 4, 5]
3 in nums
9 in nums
min(nums)
max(nums)
nums = nums + [9, 10]
print(nums)
nums[0]
nums[2]
nums[-1]
nums[-2]
nums[2:5]
len(nums)
nums[2] = 11
print(nums)
nums[2:6] = [20,30]
```

Note: Lists are mutable while strings aren't!