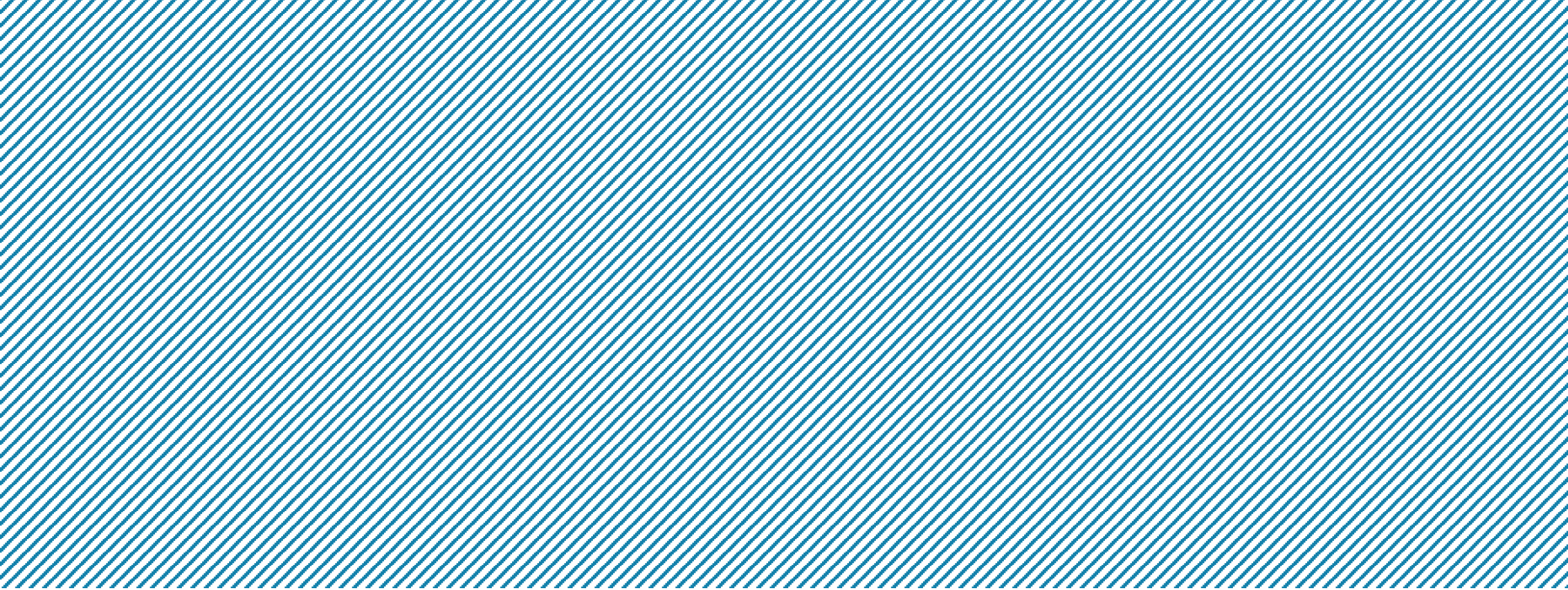
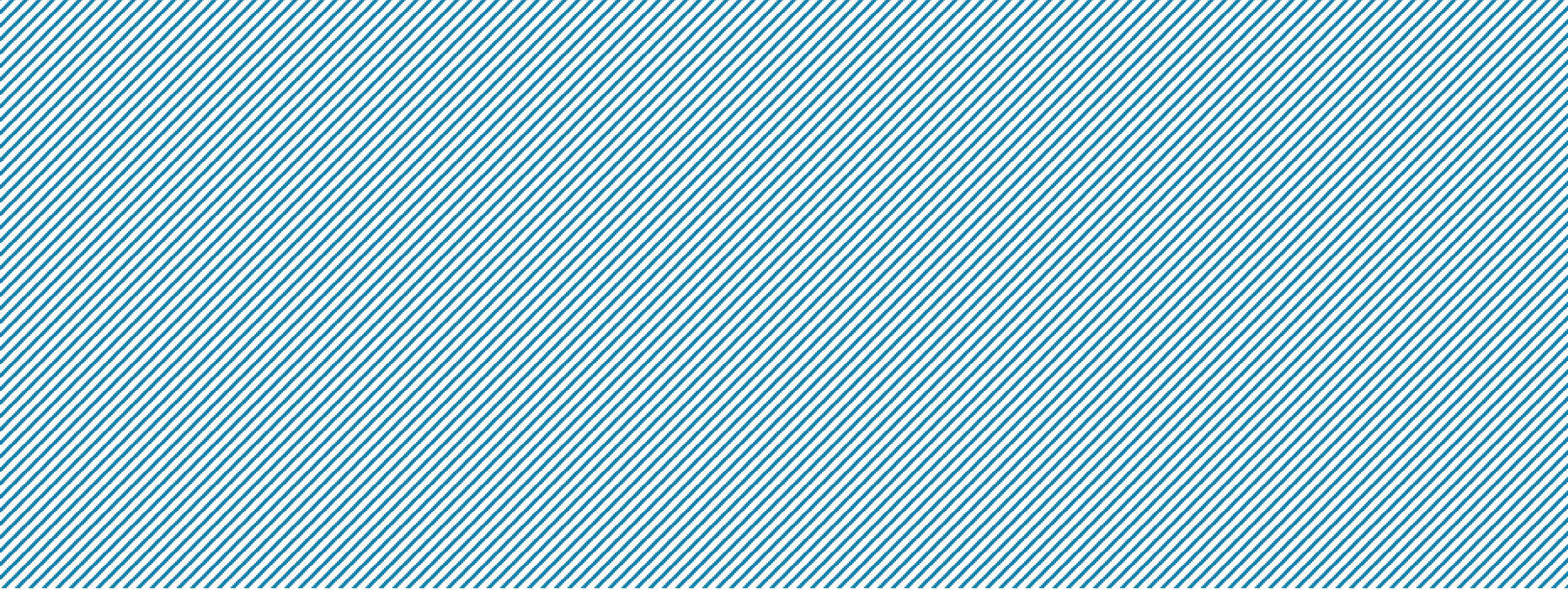


PROGRAMMING LANGUAGES PARADIGMS

Chapter 4: Python



STRINGS

STRINGS

```
size = len('I love math!')
print (size)        #will print 12
```

Python's **len** function returns the the length of a string

```
name = "John Smith"
print (name[0])                 #prints 'J'
print (name[len(name)-1])       #prints 'h'
```

Indexing the symbols in a string: we can find the symbol/letter at any given position or index.

```
bestFood = 'bacon cheeseburger'
print(bestFood[0:5])            #prints "bacon"

bestFood = 'bacon cheeseburger'
print(bestFood[6:12])           #prints "cheese"
```

Slicing a string: We can find parts of a string using its special slicing notation.

```
name = "John Smith"
for letter in name:
    print (letter) #prints one letter per line
```

You can use an enhanced for loop to go through all the symbols/letters in a string, processing one at a time.

STRING ARITHMETIC

- Strings can be “added” together. Adding two strings results in a new string that is simply the first one followed by the second. This is called string concatenation.

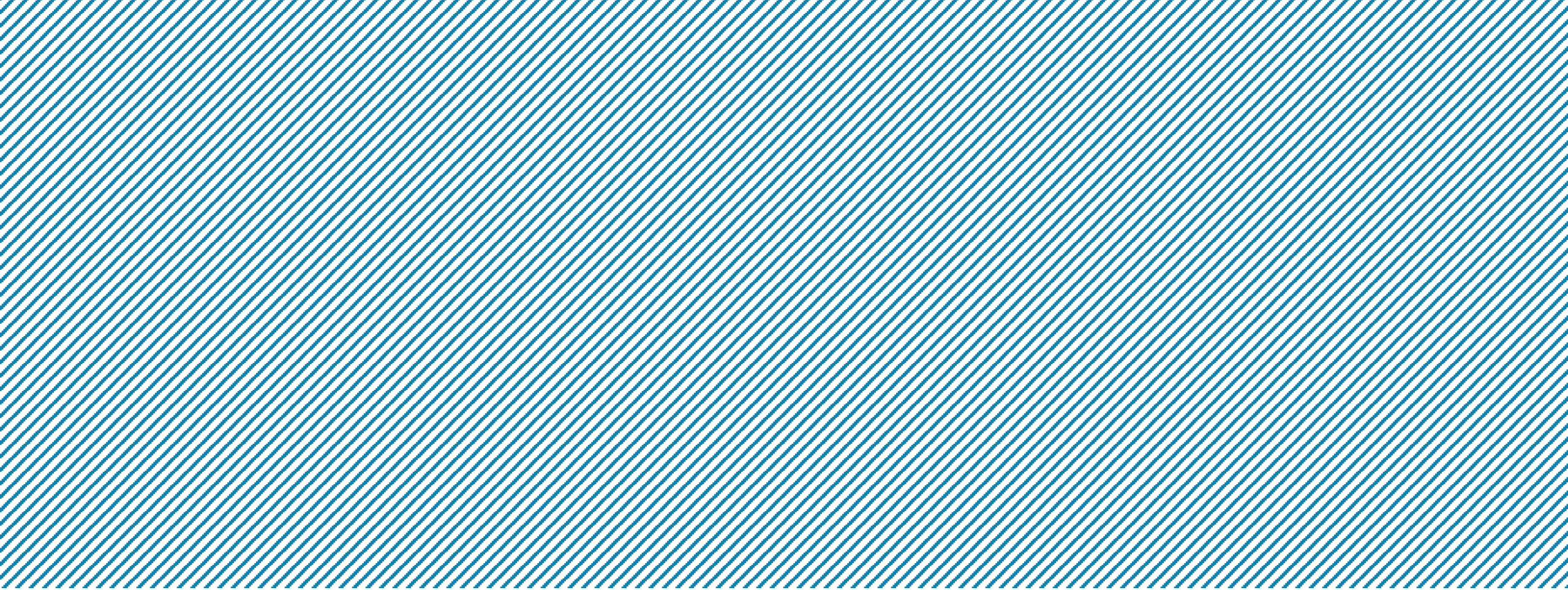
```
bestFood = "bacon " + "cheeseburger"
print(bestFood)              → prints "bacon cheeseburger"

name = "John"
name = name + "Smith"
print(name)                  → prints "John Smith"
```

- Strings can also be “multiplied”

```
meat = "bacon"
print (meat * 3)             → prints "bacon bacon bacon"
```

STRING COMPARISON

- Given the following variable assignments:
  - name1 = ”John Smith”
  - name2 = ”Smith”
  - name3 = ”John Smith”
- We can make the following comparisons, for example:
  - name1 == name3 → **true**
  - name2 < name1 → **false** **(alphabetical order)**



LISTS

CREATING A LIST

```
oddNumbers = [1 , 3 , 5 , 7 , 9 , 11]

friends = ["Rachel”, "Monica", "Phoebe", "Joey", "Ross", "Chandler"]
```

LISTS AND STRINGS INDEXING

- We can find the symbol/letter at any given position in a string (the first symbol in a string has index 0)

```
name = “John Smith”
name[1]          → 1st letter in name (‘o’)
name[5]          → 5th letter in name (‘S’)
name[n]          → nth  letter in name
```

- We can find the item at any given position in a list (the first symbol in a list has index 0)

```
friends = ["Rachel", "Monica", "Phoebe", "Joey", "Ross", "Chandler"]
friends[1]       → 1st item in friends (‘Monica’)
friends[4]       → 4th item in friends (‘Ross’)
friends[n]       → nth  item in friends
```

LISTS AND STRINGS INDEXING

- We can find the symbol/letter at any given position in a string (the first symbol in a string has index 0)

  name = “John Smith”

  name[1] → 1st letter in name (‘o’)

  name[5] → 5th letter in name (‘S’)

  name[n] → nth letter in name

- We can find the item at any given position in a list (the first symbol in a list has index 0)

  friends = ["Rachel", "Monica", "Phoebe", "Joey", "Ross", "Chandler"]

  friends[1] → 1st item in friends (‘Monica’)

  friends[4] → 4th item in friends (‘Ross’)

  friends[n] → nth item in friends

LISTS: LENGTH, INDEXING, AND SLICING

- Almost everything that works on strings also works on lists.

```
oddNumbers = [1 , 3 , 5 , 7 , 9 , 11]
friends = ["Rachel", "Monica", "Phoebe", "Joey", "Ross", "Chandler"]
```

- Length

  print(len(friends)) → will print **6**

- Indexing

  print(friends[0]) → will print **Rachel**

  print(friends[2]) → will print **Phoebe**

- Slicing

  print (friends[0:3]) → will print **['Rachel', 'Monica', 'Phoebe']**

LISTS ARITHMETIC

- Just like strings, lists can be added as well.
  - Adding two lists together creates a new list that contains all of the elements in the first list followed by all of the elements in the second.
  - This is called list concatenation and is similar to string concatenation.

```
mylist = [1, 2, 3]
yourlist = [9, 8, 7]
ourlists = mylist + yourlist
print(ourlists)              → will print [1,2,3,9,8,7]

newlist = mylist + [4]
print(newlist)               → will print [1,2,3,4]
```

- Lists can also be "*multiplied*"

```
repeatedList = mylist * 3
print (repeatedList)    → will print [1,2,3,1,2,3,1,2,3]
```

GOING THROUGH EACH ITEM IN A LIST: *ENHANCED FOR LOOP*

- You can use an *enhanced for loop* to go through all the items in a list, processing one at a time.

```
friends = [ "Rachel" , "Monica" , "Phoebe", "Ross" "Chandler", "Joey"]
for item in friends:
    print (item)
```

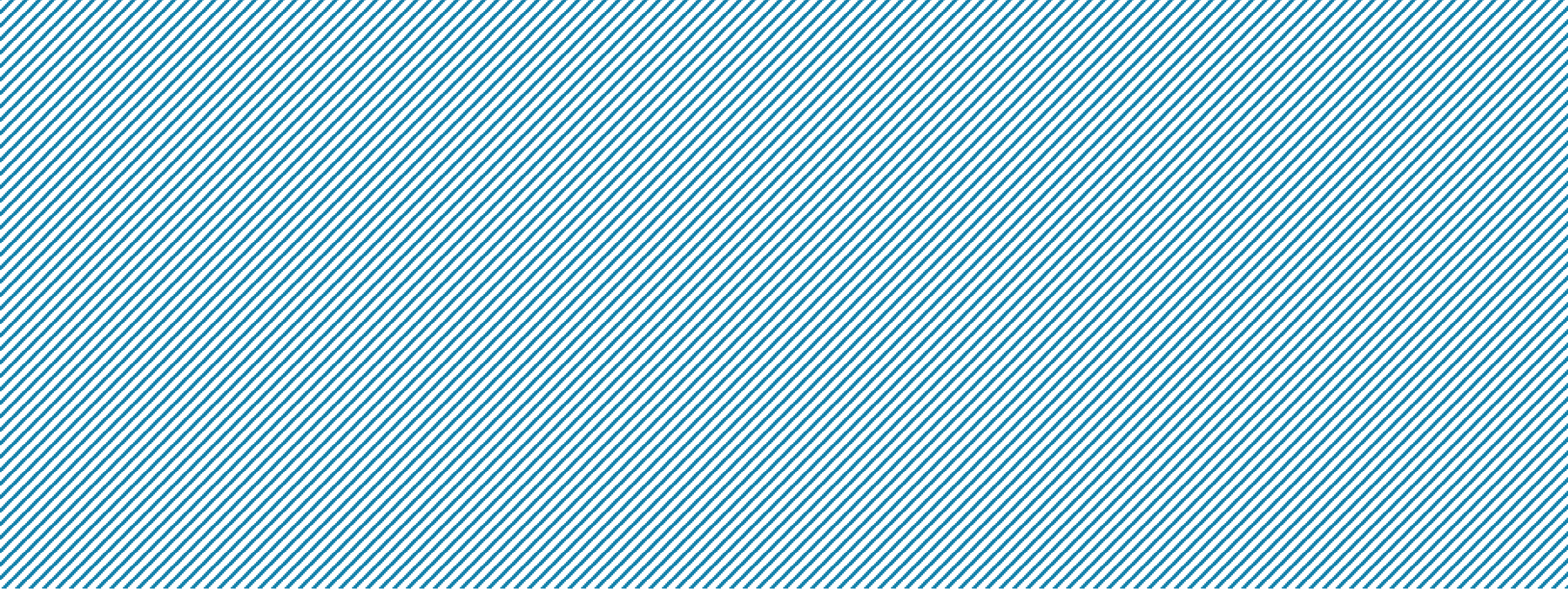
Rachel
Monica
Phoebe
Ross
Chandler
Joey

GOING THROUGH EACH ITEM IN A LIST: *ENHANCED FOR LOOP*

- You can use an enhanced for loop to go through all the items in a list, processing one at a time.

```
numbers = [2, 4, 8, 16, 32, 64]
sum = 0
for item in numbers:
    sum = sum + item
print ("The sum is: ", sum)
```

The sum is: 126



DICTIONARIES

DICTIONARIES

```
states = {'AL': 'Alabama', 'AK': 'Alaska', 'CA': 'California',
          'FL': 'Florida', 'HI': 'Hawaii', 'MD': 'Maryland',
          'NJ': 'New Jersey', 'NY': 'New York', 'TX': 'Texas'}
```

- In a dictionary, a key and its value are separated by a colon.
- The **key, value** pairs are separated with commas.
- The full list of **key, value** pairs is enclosed between curly brackets " { } "

DICTIONARIES: RETRIEVING THE VALUE ASSOCIATED WITH A KEY

```
states = {'AL': 'Alabama', 'AK': 'Alaska', 'CA': 'California',
          'FL': 'Florida', 'HI': 'Hawaii', 'MD': 'Maryland',
          'NJ': 'New Jersey', 'NY': 'New York', 'TX': 'Texas'}

months = { 1: "January", 2 : "February",  3 : "March",
           4 : "April", 5 : "May", 6 : "June", 7 : "July",
           8 : "August", 9 : "September", 10 : "October",
           11 : "November", 12 : "December" }
```

- To get a value out of a dictionary, you must supply its key. We do that using square brackets "[ ]"

  months[1] → "January"

  months[3] → "March"

  states["FL"] → "Florida"

  states["HI"] → "Hawaii"

DICTIONARIES: RETRIEVING THE LIST OF KEYS

```
months = { 1: "January", 2 : "February",  3 : "March",
           4 : "April", 5 : "May", 6 : "June", 7 : "July",
           8 : "August", 9 : "September", 10 : "October",
           11 : "November", 12 : "December" }
```

- Performing **`list(d.keys())`** on a dictionary (where d is the name of the dictionary) returns a list of all the keys used in the dictionary.

```
print ("The dictionary contains the following keys: ", list(months.keys()))
```

```
The dictionary contains the following keys:  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

DICTIONARIES: CHECKING WHETHER A GIVEN KEY IS IN THE DICTIONARY

- To check whether a given key is in the dictionary, use the keyword **in**.

```
states = {'AL': 'Alabama', 'AK': 'Alaska', 'CA': 'California',
          'FL': 'Florida', 'HI': 'Hawaii', 'MD': 'Maryland',
          'NJ': 'New Jersey', 'NY': 'New York', 'TX': 'Texas'}

state = input("Enter a state abbreviation:")
if state in states:
    print(states[state])
else:
    print("I don't have that state in my list")
```

DICTIONARIES: ITERATING OVER THE KEYS OF A DICTIONARY; *ENHANCED FOR LOOP*

```
months = { 1: "January", 2 : "February",  3 : "March",
           4 : "April", 5 : "May", 6 : "June", 7 : "July",
           8 : "August", 9 : "September", 10 : "October",
           11 : "November", 12 : "December" }
for key in months:
    print (key, months[key])
```

1 January
2 February
3 March
4 April
5 May
6 June
7 July
8 August
9 Setember
10 October
11 November
12 December

COMBINING DICTIONARIES AND LISTS

- In the following examples, the value associated with each key is a **list:**

KEY	VALUE
Spring	[March, April, May, June]
Summer	[June, July, August, September]
Fall	[September, October, November, December]
Winter	[December, January, February, March]

```
seasons = {"Spring": ["March", "April", "June"],
           "Summer": ["June", "July", "August", "September"]
               ... }
spring_months = seasons["Spring"]
print(spring_months)
print(spring_months[2])
print(seasons["Spring"][2])
```

→ [March, April, May, June]

→ May

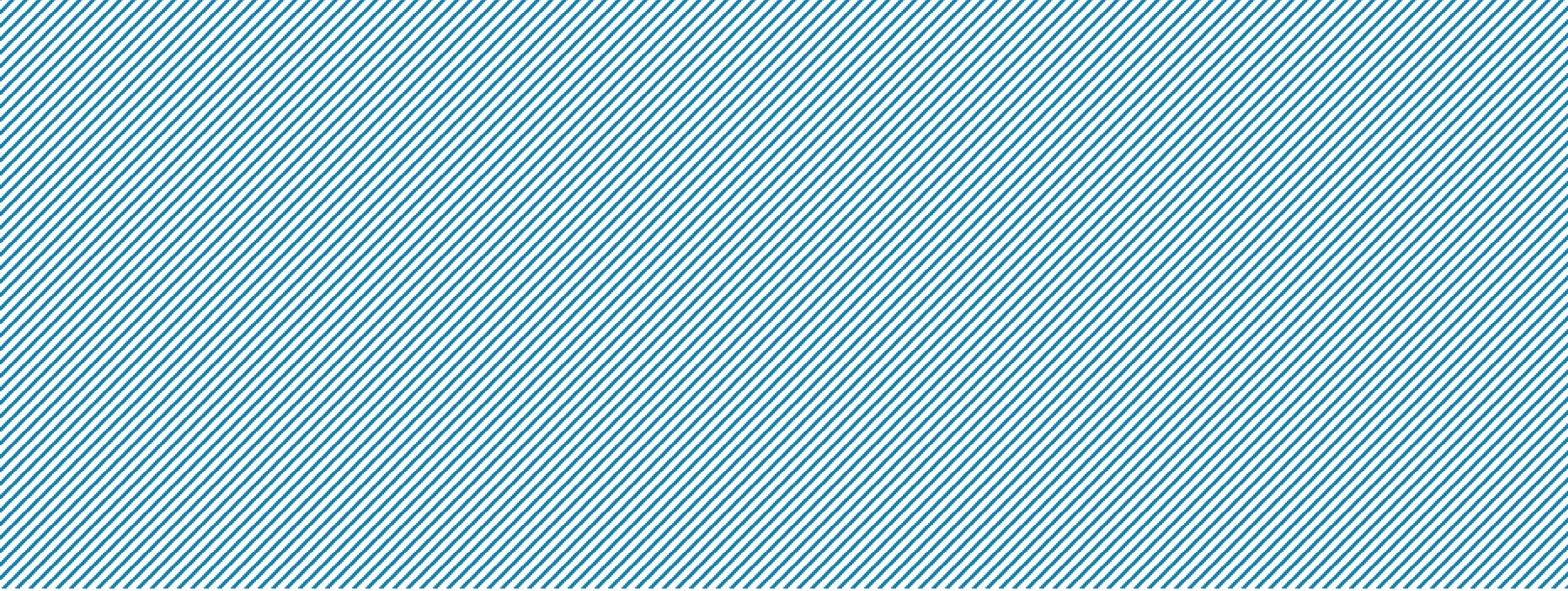
→ May

LENGTH OF A DICTIONARY

```
states = {'AL': 'Alabama', 'AK': 'Alaska', 'CA': 'California',
          'FL': 'Florida', 'HI': 'Hawaii', 'MD': 'Maryland',
          'NJ': 'New Jersey', 'NY': 'New York', 'TX': 'Texas'}
months = { 1: "January", 2 : "February",  3 : "March",
           4 : "April", 5 : "May", 6 : "June", 7 : "July",
           8 : "August", 9 : "September", 10 : "October",
           11 : "November", 12 : "December" }
```

- We can find the length of a dictionary by using Python's **len** function:

```
len(states)    → 9
len(months)    → 12
```



LIST COMPREHENSIONS, ITERABLES AND GENERATORS

LIST COMPREHENSIONS

- List comprehensions provide a concise way to create lists.
- Consist of brackets containing an expression followed by a **for** clause, then zero or more **for** or **if** clauses.
- The result will be a list resulting from evaluating the expression in the context of the **for** and **if** clauses which follow it.
- Example:

```
x = [i for i in range(10)]

print (x)
```

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

More about list comprehensions: https://www.pythonforbeginners.com/basics/list-comprehensions-in-python

FOR LOOP VS LIST COMPREHENSIONS

```
# You can either use loops:
squares = []
for x in range(10):
    squares.append(x**2)
print (squares)
```

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

```
# Or you can use list comprehensions to get the same result:
squares = [x**2 for x in range(10)]
print (squares)
```

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

FOR LOOP VS LIST COMPREHENSIONS

```
# You can either use loops:
even = []
for x in range(10):
    if (x%2 == 0):
        even.append(x)
print (even)
```

```
[ 0, 2, 4, 6, 8 ]
```

```
# Or you can use list comprehensions to get the same result:
even = [x for x in range(10) if x%2 == 0]
print (even)
```

```
[ 0, 2, 4, 6, 8 ]
```

LIST COMPREHENSION IN PYTHON: THE MATHEMATICS

- In Math, the common ways to describe lists (or sets, or tuples, or vectors) are:

$S = \{x^2 : x \text{ in } \{0 \ldots 9\}\}$
$V = (1, 2, 4, 8, \ldots, 2^{12})$
$M = \{x \mid x \text{ in } S \text{ and } x \text{ even}\}$

- The actual lists that these definitions would produce are:

S = {0, 1, 4, 9, 16, 25, 36, 49, 64, 81}
V = {1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096}
M = {0, 4, 16, 36, 64}

- Python List Comprehensions:

```
S = [x**2 for x in range(10)]
V = [2**i for i in range(13)]
M = [x for x in S if x % 2 == 0]
```

EXERCISES

- What is the output of the following code?

```
numbers = range(12)
new_list = []
for n in numbers:
    if n%2==0:
        new_list.append(n**2)
print(new_list)
```

- Write equivalent code using a list comprehension
- Write a list comprehension to convert the following list from km to feet (there are 3280.8399 feet in a km)

```
kilometers = [3.2, 5, 6.6, 8, 10, 22, 44]
```

- What is the output of the following code?

```
divided = []
for x in range(100):
    if x%2 == 0 :
        if x%6 == 0:
            divided.append(x)
```

- Write equivalent code using a list comprehension

EXERCISES

- What is the output of the following code?

```
feet = [128608, 119750, 122375, 124015]
new_list = []
for x in feet:
    if x >= 120000:
        new_list.append(x+1)
    else:
        new_list.append(x+5)
print(new_list)
```

- Write equivalent code using a list comprehension

NESTED LIST COMPREHENSIONS

```
list_of_list = [[1,2,3],[4,5,6],[7,8][~ja]

# Flatten list_of_list
Flat = [y for x in list_of_list for y in x]
print(Flat)
```

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

```
list_of_list = [[1,2,3],[4,5,6],[7,8],["hello", "goodbye"]]

# Flatten list_of_list
Flat = [y for x in list_of_list for y in x]
print(Flat)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 'hello', 'goodbye']
```

```
Pairs = [(x,y) for x in range(3) for y in range(4)]
print (Pairs)
```

```
[(0, 0), (0, 1), (0, 2), (0, 3), (1, 0), (1, 1), (1, 2), (1, 3), (2, 0), (2, 1), (2, 2), (2, 3)]
```

https://www.geeksforgeeks.org/nested-list-comprehensions-in-python/

ITERABLES

- An iterable is an object that produces a sequence of values. Lists, tuples, sets, dictionaries, and objects of a few other built-in types are iterables.
- Iterables can be used in for statements and a number of built-in functions, including min, max, sum, all, any, filter, map, sorted, and zip.

```
for x in (1,2,3): print(x)          # elements of a tuple
for x in [1,2,3]: print(x)          # elements of a list
for x in {1,2,3}: print(x)          # elements of a set
for c in 'hello': print(c)          # characters of a string

for k in {'x':1, 'y':2, 'z':3}:
    print(k)                        # keys of a dict

with open('colors') as f:
    for line in f:                  # lines of a file
        print(line.strip())
```

GENERATORS

- A generator is a function that returns a generator object which we can iterate over (one value at a time).
- A generator results from calling a function containing a **yield** statement.
- Executing such a function does not invoke the function's body, but rather **returns an iterator object** (just an object we can iterate over).
- Here's a generator that produces successive powers of two, up to some limit:

```
def powers_of_two(limit):
    value = 1
    while value < limit:
        yield value
        value += value

for i in powers_of_two(40):
    print(i)
```

```
1
2
4
8
16
32
```

yield may be called with a value, in which case that value is treated as the "generated" value.

The next time **next()** is called on the generator, the generator resumes execution from where it called **yield**, not from the beginning of the function.

All of the state, like the values of local variables, is recovered and the generator continues to execute until the next call to **yield**.

GENERATORS

- A generator is a function that returns a generator object which we can iterate over (one value at a time).
- A generator results from calling a function containing a **yield** statement.
- Executing such a function does not invoke the function's body, but rather **returns an iterator object** (just an object we can iterate over).
- Here's a generator that produces even numbers, starting at an upper value and down to 2:

```
def even(x):
    while(x!=0):
        if x%2==0:
            yield x
        x-=1

e=even(10)
print(next(e))
print(next(e))
print(next(e))
print(next(e))
```

```
10
8
6
4
```

yield may be called with a value, in which case that value is treated as the "generated" value.

The next time **next()** is called on the generator, the generator resumes execution from where it called **yield**, not from the beginning of the function.

All of the state, like the values of local variables, is recovered and the generator continues to execute until the next call to **yield**.

GENERATORS

- If the iteration scheme is sufficiently simple, you can create a generator with a ***generator expression***, which looks like a *list comprehension* with parentheses instead of square brackets.
- Generator expressions have the advantage over list comprehensions of not having to compute and store the entire data set in memory.

```
# List comprehension.
# Computes all of its elements first.
# If large, it's too slow to produce and wastes memory.
bad = [x*x for x in range(10**8)]
print("DONE 1")
print(len(bad))
```

```
# Generator expression.
# Produces values on demand, during iteration.
# Computed instantly and consumes almost no memory.
good = (x*x for x in range(10**9))

print(next(good))
print(next(good))
```

THE MODULE **ITERTOOLS**

- The module **itertools** from the standard library includes a number of functions for constructing and manipulating *generators*, including permutations, which builds a generator that produces permutations of a sequence as tuples.

```
import sys
from itertools import permutations
if len(sys.argv) != 2:
    sys.stderr.write('Exactly one argument is required\n')
    sys.exit(1)

for word in (''.join(p) for p in permutations(sys.argv[1])):
    print(word)
```

```
$ python permutations.py rat
rat
rta
art
atr
tra
tar
```

More about generators: https://realpython.com/introduction-to-python-generators