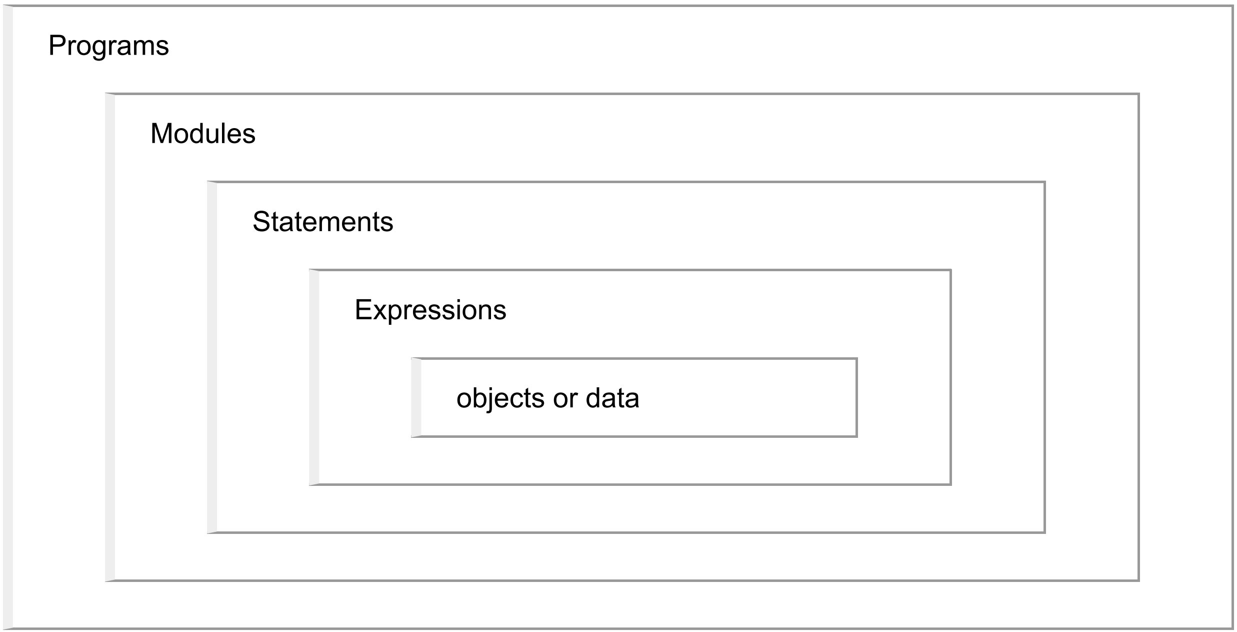
Overview of Python Object Types

- in a Python script we consider everything as an object.
- data is also considered as an objects, so object type is nothing but an datatypes.
- there are mainly two data types in python...
  1. built-in objects that Python provides.(primitive datatype)
  2. objects we create using Python classes or external language tools such as C extension libraries. (non-primitive datatype)
     - also known as reference type

Python Conceptual Hierarchy



here we can clearly see that...

- data is object and by Expression we process on data.
- if we combine many Expressions and data it brecomes a sentenece.
- module contains a many statements and program is composed of modules.

Why Use Built-in Types?

- Built-in objects make programs easy to write.
- Built-in objects are components of extensions and reference or non primitive datatypes.
  - objects implemented manually are often built on top of built-in types such as lists and dictionaries. For instance, a stack data structure may be implemented as a class that manages or customizes a built-in list.
- Built-in objects are often more efficient than custom data structures.

Built-in objects preview

Object type	Example literals/creation
Numbers	1234, 3.1415, 3+4j, 0b111, Decimal(), Fraction()
Strings	'spam', "Bob's", b'a\x01c', u'sp\xc4m'
Lists	[1, [2, 'three'], 4.5], list(range(10))
Dictionaries	{'food': 'spam', 'taste': 'yum'}, dict(hours=10)
Tuples	(1, 'spam', 4, 'U'), tuple('spam'), namedtuple
Files	open('eggs.txt'), open(r'C:\ham.bin', 'wb')
Sets	set('abc'), {'a', 'b', 'c'}
Other core types	Booleans, types, None,Fraction,decimal,class

Numbers

- integers that have no fractional part
- floating-point numbers that do and more exotic types
- complex numbers with imaginary parts
- decimals with fixed precision
- rationals with numerator and denominator and full-featured sets.

Numbers in Python support the normal mathematical operations. for Ex...

- the plus sign `+` performs addition, a star `*` is used for multiplication, and two stars `**` are used for exponentiation

In [1]:

```
123 + 222 # Integer addition
```

Out[1]:

```
345
```

In [2]:

```
1.5 * 4 # Floating-point multiplication
```

Out[2]:

```
6.0
```

In [3]:

```
2 ** 100 # 2 to the power 100
```

Out[3]:

```
1267650600228229401496703205376
```

In [4]:

```
# number object has no len object so it wouldn't work here...
len(str(2 ** 10000)) # How many digits in a really BIG number?
```

Out[4]:

```
3011
```

3.1415 * 2 # repr: as code (Pythons less than 2.7 and 3.1)
>>> 6.2830000000000004
print(3.1415 * 2) # str: user-friendly
>>> 6.283

- first form is known as an object's as-code repr
- second is known as a its user-friendly str

Strings

- Strings are used to record both textual information (your name, for instance) as well as arbitrary collections of bytes.
- strings are sequences of one-character strings.
- more general sequence types include lists and tuples

In Python, indexes are coded as offsets from the front, and so start from 0: the first item is at index 0, the second is at index 1, and so on

Sequence Operations

- we can verify string length with the built-in len function
- we can fetch string components with indexing expressions

In [5]:

```
S = 'Spam' # Make a 4-character string, and assign it to a name
len(S) # Length
```

Out[5]:

```
4
```

In [6]:

```
S[0] # The first item in S, indexing by zero-based position
```

Out[6]:

```
'S'
```

In [7]:

```
S[1] # The second item from the left
```

Out[7]:

```
'p'
```

- In Python we can also index backward from the end
    - positive indexes count from the left
    - negative indexes count back from the right

In [8]:

```
S[-1] # The last item from the end in S
```

Out[8]:

```
'm'
```

In [9]:

```
S[-2] # The second-to-last item from the end
```

Out[9]:

```
'a'
```

- technically a negative index is simply added to the string's length so the following two operations are equivalent

In [10]:

```
S[-1] # The last item in S
```

Out[10]:

```
'm'
```

In [11]:

```
S[len(S)-1] # Negative indexing, the hard way
```

Out[11]:

```
'm'
```

- sequences also support a more general form of indexing known as slicing.
- which is a way to extract an entire section (slice) in a single step. For example:

In [12]:

```
S
```

Out[12]:

```
'Spam'
```

In [13]:

```
# it Slice of S from offsets 1 to offsets 3. here...
# first offset is included and last offset is excluded
S[1:3]
```

Out[13]:

```
'pa'
```

- general form of slicing is X[I:J]
- it returns everything in X from offset I up to offset J but not including offset J.
- The result is returned in a new object.

- In a slicing....
  - the left bound defaults to zero
  - the right bound defaults to the length of the sequence being sliced.

In [14]:

```
S[1:] # Everything past the first (1:len(S))
```

Out[14]:

```
'pam'
```

In [15]:

```
S # S itself hasn't changed
```

Out[15]:

```
'Spam'
```

In [16]:

```
S[0:3] # Everything but the last
```

Out[16]:

```
'Spa'
```

In [17]:

```
S[:3] # Same as S[0:3]
```

Out[17]:

```
'Spa'
```

In [18]:

```
S[:-1] # Everything but the last again, but simpler (0:-1)
```

Out[18]:

```
'Spa'
```

In [19]:

```
S[:] # All of S as a top-level copy (0:len(S))
```

Out[19]:

```
'Spam'
```

- as sequences strings also support concatenation with the plus sign and repetition with star sign.
- concatenation means joining two strings into a new string.
- repetition means making a new string by repeating string many times.

In [20]:

```
S
```

Out[20]:

```
'Spam'
```

In [21]:

```
S + ' HITESH' # Concatenation
```

Out[21]:

```
'Spam HITESH'
```

In [22]:

```
S # original string S is unchanged as it creates a new object each time.
```

Out[22]:

```
'Spam'
```

In [23]:

```
# if we want to make change in original string and modify it we need to assign it with o
# howevwer this operatiion also cretes a new object.
S = S + ' HITESH ' # Concatenation
```

In [24]:

```
S
```

Out[24]:

```
'Spam HITESH '
```

In [25]:

```
>>> S * 8 # Repetition
```

Out[25]:

```
'Spam HITESH Spam HITESH Spam HITESH Spam HITESH Spam HITESH Spam HITESH S
pam HITESH Spam HITESH '
```

in Python...

- the meaning of an operation depends on the objects being operated on.
- the plus sign (+) means different things for different objects: addition for numbers, and concatenation for strings.
- This property of Python is called as a polymorphism.

Immutability

- strings are immutable datatype.
- we can never overwrite the values of immutable objects.
- beacause of this we were not changing the original string with any of the operations we ran on it.
- Every string operation is defined to produce a new string as its result.

In [26]:

```
S = 'Spam'
```

In [27]:

```
S[0]
```

Out[27]:

```
'S'
```

In [28]:

```
S[0] = 'z' # Immutable objects cannot be changed
```

```
---------------------------------------------------------------------------
-
TypeError                                 Traceback (most recent call las
t)
Input In [28], in <cell line: 1>()
----> 1 S[0] = 'z'

TypeError: 'str' object does not support item assignment
```

In [29]:

```
 # But we can run expressions to make new objects
S = 'z' + S[1:]
```

In [30]:

```
S
```

Out[30]:

```
'zpam'
```

- Every object in Python is classified as either immutable (unchangeable) or not.
- In terms of the core types...
  - numbers, strings, and tuples are immutable.
  - lists, dictionaries, and sets are not-immutable.

- we can change text-based data in place by...
  - expand it into a list of individual characters and join it back together with nothing between.
  - using the newer bytearray type available in Python.

In [31]:

```
S = 'HITEDH'
L = list(S) # Expand to a list: [...]
L
```

Out[31]:

```
['H', 'I', 'T', 'E', 'D', 'H']
```

In [32]:

```
L[4] = 'S' # Change it in place
S = ''.join(L) # Join with empty delimiter
S
```

Out[32]:

```
'HITESH'
```

In [33]:

```
B = bytearray(b'spam')
B.extend(b'eggs')
B
```

Out[33]:

```
bytearray(b'spameggs')
```

In [34]:

```
B.decode() # Translate to normal string
```

Out[34]:

```
'spameggs'
```

- string find method is the basic substring search operation in string.
- it returns the offset of the passed-in substring or −1 if it is not present.

- it always returns the first occurence of the string matched.
- string replace method performs global searches and replacements

In [35]:

```
S = 'Hi my name is Hitesh'
>>> S.find('Hi') # Find the offset of a substring in S
```

Out[35]:

```
0
```

In [36]:

```
S
```

Out[36]:

```
'Hi my name is Hitesh'
```

In [37]:

```
S.replace('Hi', 'mine') # Replace occurrences of a string in S with another
```

Out[37]:

```
'mine my name is minetesh'
```

In [38]:

```
S
```

Out[38]:

```
'Hi my name is Hitesh'
```

- split method - split a string into substrings based on a delimiter
- Upper() and Lower() - perform case conversions
- isalpha(), isnum() - test the content of the string (digits,letters, and so on)
- strip()- removes a whitespace characters off the ends of the string.

In [39]:

```
line = 'aaa,bbb,ccccc,dd'
line.split(',') # Split on a delimiter into a list of substrings
```

Out[39]:

```
['aaa', 'bbb', 'ccccc', 'dd']
```

In [40]:

```
S = 'spam'
S.upper() # Upper- and Lowercase conversions
```

Out[40]:

```
'SPAM'
```

In [41]:

```
S = 'SPAM'
S.lower() # Upper- and lowercase conversions
```

Out[41]:

```
'spam'
```

In [42]:

```
S.isalpha() # Content tests: isalpha, isdigit, etc.
```

Out[42]:

```
True
```

In [43]:

```
line = 'aaa,bbb,ccccc,dd\n'
line.rstrip() # Remove whitespace characters on the right side
```

Out[43]:

```
'aaa,bbb,ccccc,dd'
```

In [44]:

```
line = '\n\naaa,bbb,ccccc,dd\n'
line.lstrip() # Remove whitespace characters on the right side
```

Out[44]:

```
'aaa,bbb,ccccc,dd\n'
```

In [45]:

```
line = 'aaa,bbb,ccccc,dd\n'
line.strip() # Remove whitespace characters on the right side
```

Out[45]:

```
'aaa,bbb,ccccc,dd'
```

In [46]:

```
''.join(line.rstrip().split(',')) # Combine two or more operations
```

Out[46]:

```
'aaabbbcccccdd'
```

In [47]:

```
''.join(line.split(',')).replace('\n','')  # Combine two or more operations
```

Out[47]:

```
'aaabbbcccccdd'
```

here...it strips before it splits because Python runs from left to right making a temporary result along the way.

- Strings also support an advanced substitution operation known as formatting.

In [48]:

```
'{0}, Reshma, and {1}'.format('Hitesh', 'Purvi!') # Formatting method (numbers are compu
```

Out[48]:

```
'Hitesh, Reshma, and Purvi!'
```

In [49]:

```
# Numbers optional in python 2.7+, 3.1+
'{}, Reshma, and {}'.format('Hitesh', 'purvi!')
```

Out[49]:

```
'Hitesh, Reshma, and purvi!'
```

In [50]:

```
# formating also supported in nmubers as well
'{:,.2f}'.format(296999.2567)
```

Out[50]:

```
'296,999.26'
```

In [51]:

```
'{:,.6f}'.format(296999.2567)
```

Out[51]:

```
'296,999.256700'
```

In [52]:

```
# Getting Help - Assuming S is a string
print(dir(S))
```

```
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc_
_', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__',
'__getnewargs__', '__gt__', '__hash__', '__init__', '__init_subclass__',
'__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul_
_', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'capitaliz
e', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'fi
nd', 'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isascii', 'is
decimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintabl
e', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip',
'maketrans', 'partition', 'removeprefix', 'removesuffix', 'replace', 'rfin
d', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitli
nes', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'z
fill']
```

- The `__add__` method of strings performs concatenation.

In [53]:

```
S = 'Hitesh'
S + ' Vaghela'
```

Out[53]:

```
'Hitesh Vaghela'
```

In [54]:

```
S.__add__(' Vaghela')
```

Out[54]:

```
'Hitesh Vaghela'
```

- leading and trailing double underscores is the naming pattern Python uses for implementation details.
- The names without the underscores in this list are the callable methods on string objects.
- The dir function simply gives the methods' names.
- To ask what that methods do we can pass them to the help function

In [55]:

```
help(S.replace)
```

```
Help on built-in function replace:

replace(old, new, count=-1, /) method of builtins.str instance
    Return a copy with all occurrences of substring old replaced by new.

      count
        Maximum number of occurrences to replace.
        -1 (the default value) means replace all occurrences.

    If the optional argument count is given, only the first count occurren
ces are
    replaced.
```

- string method ord used to convert a character and number into ASCII values.

In [56]:

```
ord('\n') # \n is a byte with the binary value 10 in ASCII
```

Out[56]:

```
10
```

In [57]:

```
ord('H')
```

Out[57]:

```
72
```

Unicode Strings

- strings also come with full Unicode support required for processing text in internationalized character sets.
- Characters in the Japanese and Russian alphabets for example are outside the ASCII set.
- Such non-ASCII text can show up in web pages, emails, GUIs, JSON, XML, or elsewhere.
- When it does, handling it well requires Unicode support.
- Python has such support built in but the form of its Unicode support varies per Python line and is one of their most prominent differences.

In [58]:

```
'sp\xc4m'
```

Out[58]:

```
'spÄm'
```

In [59]:

```
b'a\x01c' # bytes strings are byte-based data
```

Out[59]:

```
b'a\x01c'
```

Lists

- list is a one of the most used datatype in python.
- Python list object is the most general sequence provided by the language.
- Lists are positionally ordered collections.
- Lists are mutable—unlike strings therefor lists can be modified.

Sequence Operations

- Because lists are ordered, lists support all the sequence operations we discussed for strings.
- the only difference is that the results are usually lists instead of strings.

In [60]:

```
L = [123, 'spam', 1.23] # A list of three different-type objects
```

In [61]:

```
len(L) # Number of items in the list
```

Out[61]:

```
3
```

In [62]:

```
L[0] # Indexing by position
```

Out[62]:

```
123
```

In [63]:

```
 L[:-1] # Slicing a list returns a new list
```

Out[63]:

```
[123, 'spam']
```

In [64]:

```
L + [4, 5, 6] # Concat make new lists too
```

Out[64]:

```
[123, 'spam', 1.23, 4, 5, 6]
```

In [65]:

```
L
```

Out[65]:

```
[123, 'spam', 1.23]
```

In [66]:

```
L * 2 # repeat make new lists too
```

Out[66]:

```
[123, 'spam', 1.23, 123, 'spam', 1.23]
```

In [67]:

```
L # We're not changing the original list
```

Out[67]:

```
[123, 'spam', 1.23]
```

- lists have no fixed size. That is they can grow and shrink on demand, in response to list-specific operations.

In [68]:

```
L.append('NI') # Growing: add object at end of list
L
```

Out[68]:

```
[123, 'spam', 1.23, 'NI']
```

In [69]:

```
L.pop(2) # Shrinking: delete an item in the middle
```

Out[69]:

```
1.23
```

In [70]:

```
L # "del L[2]" deletes from a list too
```

Out[70]:

```
[123, 'spam', 'NI']
```

- list append method expands the list’s size and inserts an item at the end.
- the pop method (or an equivalent del statement) then removes an item at a given offset causing the list to shrink.
- insert method inserts an item at an arbitrary position
- remove method removes a given item by value
- extend method adds a multiple items at the end.

In [71]:

```
M = ['bb', 'aa', 'cc']
M.sort()
M
```

Out[71]:

```
['aa', 'bb', 'cc']
```

In [72]:

```
M.reverse()
M
```

Out[72]:

```
['cc', 'bb', 'aa']
```

Bounds Checking

- Although lists have no fixed size Python still doesn’t allow us to reference items that are not present.

- Indexing off the end of a list is always a mistake but so is assigning off the end.

In [73]:

```
L
```

Out[73]:

```
[123, 'spam', 'NI']
```

In [74]:

```
L[99]
```

```
---------------------------------------------------------------------------
-
IndexError                                Traceback (most recent call las
t)
Input In [74], in <cell line: 1>()
----> 1 L[99]

IndexError: list index out of range
```

In [75]:

```
L[99] = 1
```

```
---------------------------------------------------------------------------
-
IndexError                                Traceback (most recent call las
t)
Input In [75], in <cell line: 1>()
----> 1 L[99] = 1

IndexError: list assignment index out of range
```

- unlike C language Python reports an error when we try to grow list this way Rather than silently growing the list in response
- because C language doesn’t do as much error checking as Python.
- To grow a list we call list methods such as append instead.

Nesting

- One nice feature of Python’s core data types is that they support arbitrary nesting—we can nest them in any combination and as deeply as we like.
- For example, we can have a list that contains a dictionary which contains another list, and so on.

In [76]:

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

Out[76]:

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

In [77]:

```
M[1] # Get 2nd row
```

Out[77]:

```
[4, 5, 6]
```

In [78]:

```
M[1][2] # Get 2nd row's third item(column)
```

Out[78]:

```
6
```

Comprehensions

- Python includes a more advanced operation known as a list comprehension expression
- which turns out to be a powerful way to process structures like our matrix.

In [79]:

```
col2 = [row[1] for row in M]
col2
```

Out[79]:

```
[2, 5, 8]
```

In [80]:

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

Out[80]:

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

In [81]:

```
# List comprehensions can be more complex in practice.
[row[1] + 1 for row in M]
```

Out[81]:

```
[3, 6, 9]
```

In [82]:

```
[row[1] for row in M if row[1] % 2 == 0]
[2, 8]
```

Out[82]:

```
[2, 8]
```

In [83]:

```
[i*j for i in range(1,11) for j in range(1,11) if i==5]
```

Out[83]:

```
[5, 10, 15, 20, 25, 30, 35, 40, 45, 50]
```

- we can use list comprehensions to step over a hardcoded list of coordinates and a string

In [84]:

```
doubles = [c * 2 for c in 'spam'] # Repeat characters in a string
doubles
```

Out[84]:

```
['ss', 'pp', 'aa', 'mm']
```

In [85]:

```
A = [  [1,2,3],[4,5,6],[7,8,9]  ]
diag1 = [A[i][i] for i in [0, 1, 2]] # Collect a diagonal from matrix
diag1
```

Out[85]:

```
[1, 5, 9]
```

In [86]:

```
diag2 = [A[i][j] for i in range(3) for j in range(3) if i+j == 2] # Collect a diagonal f
diag2
```

Out[86]:

```
[3, 5, 7]
```

In [87]:

```
list(range(4))
```

Out[87]:

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

In [88]:

```
list(range(-6, 7, 2))
```

Out[88]:

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

In [89]:

```
[[x ** 2, x ** 3] for x in range(4)]
```

Out[89]:

```
[[0, 0], [1, 1], [4, 8], [9, 27]]
```

In [90]:

```
[[x, x / 2, x * 2] for x in range(-6, 7, 2) if x > 0]
```

Out[90]:

```
[[2, 1.0, 4], [4, 2.0, 8], [6, 3.0, 12]]
```

In [91]:

```
A = [  [1,2,3],[4,5,6],[7,8,9]  ]
[sum(r) for r in A]
```

Out[91]:

```
[6, 15, 24]
```

- The map built-in can do similar work, by generating the results of running items through a function one at a time and on request.

In [92]:

```
list(map(sum, A)) # Map sum over items in M
```

Out[92]:

```
[6, 15, 24]
```

- comprehension syntax can also be used to in sets and dictionaries.

In [93]:

```
{row for row in range(10)}
```

Out[93]:

```
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

In [94]:

```
{i : i*10 for i in range(3)} # Creates key/value table of row sums
```

Out[94]:

```
{0: 0, 1: 10, 2: 20}
```

In [95]:

```
[ord(x) for x in 'spaam'] # List of character ordinals
```

Out[95]:

```
[115, 112, 97, 97, 109]
```

In [96]:

```
{ord(x) for x in 'spaam'} # Sets remove duplicates
```

Out[96]:

```
{97, 109, 112, 115}
```

In [97]:

```
{x: ord(x) for x in 'spaam'} # Dictionary keys are unique
```

Out[97]:

```
{'s': 115, 'p': 112, 'a': 97, 'm': 109}
```

In [98]:

```
g = (ord(x) for x in 'spaam') # Generator of values
```

In [99]:

```
next(g)
```

Out[99]:

```
115
```

Dictionaries

- dictionaries are not sequences at all but are instead known as mappings.
- Mappings are also collections of other objects but they store objects by key instead of by relative position.
- Dictionaries is the only mapping type in Python's core objects.
- Dictionaries are also mutable.
- Dictionaries may be changed in place and can grow and shrink on demand.
- Dictionaries can't contain duplicate keys.

dictionaries are coded in curly braces `{}` and consist of a series of `key: value` pairs.

In [100]:

```
D = {'food': 'Spam', 'quantity': 4, 'color': 'pink'}
D
```

Out[100]:

```
{'food': 'Spam', 'quantity': 4, 'color': 'pink'}
```

- We can index this dictionary by key to fetch and change the keys' associated values.
- The dictionary index operation uses the same syntax as that used for sequences, but the item in the square brackets is a key not a relative position.

In [101]:

```
D['food'] # Fetch value of key 'food'
```

Out[101]:

```
'Spam'
```

In [102]:

```
D['quantity'] += 1 # Add 1 to 'quantity' value
D
```

Out[102]:

```
{'food': 'Spam', 'quantity': 5, 'color': 'pink'}
```

- starts with an empty dictionary and fills it out one key at a time.
- Unlike out-of-bounds assignments in lists, which are forbidden, assignments to new dictionary keys create those keys.

In [103]:

```
D = {}
D['name'] = 'Bob' # Create keys by assignment
D['job'] = 'dev'
D['age'] = 40
D
```

Out[103]:

```
{'name': 'Bob', 'job': 'dev', 'age': 40}
```

In [104]:

```
print(D['name'])
```

```
Bob
```

- we can also make dictionaries by passing to the dict type name either keyword arguments (a special key=value syntax in function calls), or the result of zipping together sequences of keys.

In [105]:

```
bob1 = dict(name='Bob', job='dev', age=40) # Keywords
bob1
```

Out[105]:

```
{'name': 'Bob', 'job': 'dev', 'age': 40}
```

In [106]:

```
bob2 = dict(zip(['name', 'job', 'age'], ['Bob', 'dev', 40])) # Zipping
bob2
```

Out[106]:

```
{'name': 'Bob', 'job': 'dev', 'age': 40}
```

- left-to-right order of dictionary keys is scrambled.
- Mappings are not positionally ordered
- so result is in a different order than we typed them.

Nesting

In [107]:

```
rec = {'name': {'first': 'Bob', 'last': 'Smith'},
 'jobs': ['dev', 'mgr'],
 'age': 40.5}
```

In [108]:

```
rec
```

Out[108]:

```
{'name': {'first': 'Bob', 'last': 'Smith'},
 'jobs': ['dev', 'mgr'],
 'age': 40.5}
```

In [109]:

```
rec['name'] # 'name' is a nested dictionary
```

Out[109]:

```
{'first': 'Bob', 'last': 'Smith'}
```

In [110]:

```
rec['name']['last'] # Index the nested dictionary
```

Out[110]:

```
'Smith'
```

In [111]:

```
rec['jobs'] # 'jobs' is a nested list
```

Out[111]:

```
['dev', 'mgr']
```

In [112]:

```
rec['jobs'][-1] # Index the nested list
```

Out[112]:

```
'mgr'
```

In [113]:

```
rec['jobs'].append('janitor') # Expand Bob's job description in place
```

In [114]:

```
rec
```

Out[114]:

```
{'name': {'first': 'Bob', 'last': 'Smith'},
 'jobs': ['dev', 'mgr', 'janitor'],
 'age': 40.5}
```

Missing Keys: if Tests

- For example, although we can assign to a new key to expand a dictionary, fetching a nonexistent key is still a mistake:

In [115]:

```
D = {'a': 1, 'b': 2, 'c': 3}
D
```

Out[115]:

```
{'a': 1, 'b': 2, 'c': 3}
```

In [116]:

```
D['e'] = 99 # Assigning new keys grows dictionaries
D
```

Out[116]:

```
{'a': 1, 'b': 2, 'c': 3, 'e': 99}
```

In [117]:

```
D['f'] # Referencing a nonexistent key is an error
```

```
---------------------------------------------------------------------------
-
KeyError                                  Traceback (most recent call las
t)
Input In [117], in <cell line: 1>()
----> 1 D['f']

KeyError: 'f'
```

- This is what we want—it's usually a programming error to fetch something that isn't really there.
- But in some generic programs we can't always know what keys will be present when we write our code.

In [118]:

```
'f' in D
```

Out[118]:

```
False
```

In [119]:

```
if not 'f' in D: # Python's sole selection statement
    print('missing')
```

```
missing
```

- Besides the in test there are a variety of ways to avoid accessing nonexistent keys in the dictionaries we create.
- the get method and a conditional index with a default.

In [120]:

```
value = D.get('x', "NULL") # Index but with a default
value
```

Out[120]:

```
'NULL'
```

In [121]:

```
value = D.get('x', 0) # Index but with a default
value
```

Out[121]:

```
0
```

In [122]:

```
value = D['x'] if 'x' in D else 0 # if/else expression form
value
```

Out[122]:

```
0
```

Sorting Keys: for Loops

In [123]:

```
D = {'p': 1, 'h': 2, 'r': 3}
D
```

Out[123]:

```
{'p': 1, 'h': 2, 'r': 3}
```

In [124]:

```
Ks = list(D.keys()) # Unordered keys list
Ks
```

Out[124]:

```
['p', 'h', 'r']
```

In [125]:

```
Ks.sort() # Sorted keys list
Ks
```

Out[125]:

```
['h', 'p', 'r']
```

In [126]:

```
for key in Ks: # Iterate though sorted keys
    print(key, '=>', D[key])
```

```
h => 2
p => 1
r => 3
```

- in recent versions of Python it can be done in one step with the newer sorted built-in function.
- The sorted call returns the result and sorts a variety of object types.
- in this case sorting dictionary keys automatically

In [127]:

```
D
```

Out[127]:

```
{'p': 1, 'h': 2, 'r': 3}
```

In [128]:

```
for key in sorted(D):
    print(key, '=>', D[key])
```

```
h => 2
p => 1
r => 3
```

In [129]:

```
%%time
squares = [x for x in range(10000000)]
sum(squares)
```

```
CPU times: total: 1.12 s
Wall time: 1.19 s
```

Out[129]:

```
49999995000000
```

In [130]:

```
%%time
squares = []
for x in range(10000000):
    squares.append(x )
sum(squares)
```

```
CPU times: total: 2.12 s
Wall time: 2.15 s
```

Out[130]:

```
49999995000000
```

In [131]:

```
%%time
map(sum,list(range(10000000)) )
```

```
CPU times: total: 234 ms
Wall time: 245 ms
```

Out[131]:

```
<map at 0x26dd7fc62e0>
```

In [132]:

```
%%time
filter(sum,list(range(10000000)))
```

```
CPU times: total: 297 ms
Wall time: 287 ms
```

Out[132]:

```
<filter at 0x26dd7fa2190>
```

- The list comprehension and other tools like map and filter will often run faster than a for loop.

Tuples

- The tuple object cannot be changed—tuples are sequences like lists,
- tuple are immutable, like strings.

- tuple normally coded in parentheses instead of square brackets and they support arbitrary types,

In [133]:

```
T = (1, 2, 3, 4)
len(T)
```

Out[133]:

```
4
```

In [134]:

```
T + (5, 6)
```

Out[134]:

```
(1, 2, 3, 4, 5, 6)
```

In [135]:

```
T[0]
```

Out[135]:

```
1
```

In [136]:

```
T.index(4) # 4 appears at offset 3
```

Out[136]:

```
3
```

In [137]:

```
T.count(4) # 4 appears once
```

Out[137]:

```
1
```

In [138]:

```
T[0] = 2 # Tuples are immutable
```

```
---------------------------------------------------------------------------
-
TypeError                                 Traceback (most recent call las
t)
Input In [138], in <cell line: 1>()
----> 1 T[0] = 2

TypeError: 'tuple' object does not support item assignment
```

In [139]:

```
T = (2,) + T[1:] # Make a new tuple for a new value
T
```

Out[139]:

```
(2, 2, 3, 4)
```

In [140]:

```
# the parentheses enclosing a tuple’s items can usually be omitted
T = 'spam', 3.0, [11, 22, 33]
T[1]
```

Out[140]:

```
3.0
```

In [141]:

```
T[2][1]
```

Out[141]:

```
22
```

In [142]:

```
T.append(4) # AttributeError: 'tuple' object has no attribute 'append'
```

```
---------------------------------------------------------------------------
-
AttributeError                            Traceback (most recent call las
t)
Input In [142], in <cell line: 1>()
----> 1 T.append(4)

AttributeError: 'tuple' object has no attribute 'append'
```

- If we pass a collection of objects around your program as a list, it can be changed anywhere.
- if you use a tuple, it cannot. That is, tuples provide a sort of integrity.

Files

- File objects are Python code’s main interface to external files on your computer.
- They can be used to read and write text, Excel documents, saved email messages etc. stored on our machine.
- Files are a core type in python.

In [143]:

```
f = open('data.txt', 'w') # Make a new file in write mode ('w' is write)
f.write('Hello\n') # Write strings of characters to it
```

Out[143]:

```
6
```

In [144]:

```
f.write('world\n') # Return number of items written in file
```

Out[144]:

```
6
```

In [145]:

```
f.close() # Close to flush output buffers to disk
```

In [146]:

```
f = open('data.txt') # 'r' (read) is the default processing mode
text = f.read() # Read entire file into a string
text
```

Out[146]:

```
'Hello\nworld\n'
```

In [147]:

```
print(text) # print interprets control characters
```

```
Hello
world
```

In [148]:

```
text.split() # File content is always a string
```

Out[148]:

```
['Hello', 'world']
```

- the best way to read a file today is to not read it at all—files provide an iterator that automatically reads line by line in for loops and other contexts.
- read accepts an optional maximum byte/character size
- readline reads one line at a time

In [149]:

```
for line in open('data.txt'):
    print(line)
```

```
Hello

world
```

In [150]:

```
print(dir(f))
```

```
['_CHUNK_SIZE', '__class__', '__del__', '__delattr__', '__dict__', '__dir_
_', '__doc__', '__enter__', '__eq__', '__exit__', '__format__', '__ge__',
'__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__',
'__iter__', '__le__', '__lt__', '__ne__', '__new__', '__next__', '__reduce
__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '_checkClosed', '_checkReadable', '_checkSeekable', '_
checkWritable', '_finalizing', 'buffer', 'close', 'closed', 'detach', 'enc
oding', 'errors', 'fileno', 'flush', 'isatty', 'line_buffering', 'mode',
'name', 'newlines', 'read', 'readable', 'readline', 'readlines', 'reconfig
ure', 'seek', 'seekable', 'tell', 'truncate', 'writable', 'write', 'write_
through', 'writelines']
```

In [151]:

```
help(f.readlines)
```

```
Help on built-in function readlines:

readlines(hint=-1, /) method of _io.TextIOWrapper instance
    Return a list of lines from the stream.

    hint can be specified to control the number of lines read: no more
    lines will be read if the total size (in bytes/characters) of all
    lines so far exceeds hint.
```

Other Core Types

Beyond the core types we've seen so far, there are others that may or may not qualify for membership in the category, depending on how broadly it is defined.

- Sets, for example, are a recent addition to the language that are neither mappings nor sequences;
- Sets are unordered collections of unique and immutable objects.
- we can able to create sets by calling the built-in set function or using new set literals.
- the choice of new {...} syntax for set literals makes sense.
- since, sets are much like the keys of a valueless dictionary that is the reason that set can't contain duplicate data.

In [152]:

```
X = set('spam') # Make a set with built-in function
Y = {'h', 'a', 'm'} # Make a set with set literals
X
```

Out[152]:

```
{'a', 'm', 'p', 's'}
```

In [153]:

```
Y
```

Out[153]:

```
{'a', 'h', 'm'}
```

In [154]:

```
X & Y # Intersection
```

Out[154]:

```
{'a', 'm'}
```

In [155]:

```
X | Y # Union
```

Out[155]:

```
{'a', 'h', 'm', 'p', 's'}
```

In [156]:

```
X - Y # Difference
```

Out[156]:

```
{'p', 's'}
```

In [157]:

```
{n ** 2 for n in [1, 2, 3, 4]} # Set comprehension
```

Out[157]:

```
{1, 4, 9, 16}
```

- sets useful for common tasks such as...
  - filtering out duplicates
  - isolating differences
  - performing order-neutral equality tests without sorting—in lists, strings, and all other iterable objects

In [158]:

```
list(set([1, 2, 1, 3, 1])) # Filtering out duplicates (possibly reordered)
```

Out[158]:

```
[1, 2, 3]
```

In [159]:

```
set('spam') - set('ham') # Finding differences in collections
```

Out[159]:

```
{'p', 's'}
```

In [160]:

```
set('spam') == set('asmp') # Order-neutral equality tests (== is False)
```

Out[160]:

```
True
```

Sets also support in membership tests likeother collection types in Python.

In [161]:

```
'p' in set('spam'), 'p' in 'spam', 'ham' in ['eggs', 'spam', 'ham']
```

Out[161]:

```
(True, True, True)
```

- Python recently grew a few new numeric types
  1. decimal numbers - which are fixed-precision floating-point numbers.
  2. fraction numbers - which are rational numbers with both a numerator and a denominator.

In [162]:

```
import decimal
d = decimal.Decimal('3.141')
d + 1
```

Out[162]:

```
Decimal('4.141')
```

In [163]:

```
from fractions import Fraction
f = Fraction(2, 3)
f + 1
```

Out[163]:

```
Fraction(5, 3)
```

In [164]:

```
f + Fraction(1, 2) # 2/3 + 1/2 = 7/6
Fraction(7, 6)
```

Out[164]:

```
Fraction(7, 6)
```

Boolean

- Boolean is also a python's core datatype.
- Boolean has a predefined True and False objects that are essentially just the integers 1 and 0 with custom display logic.
- Boolean also has a long supported special placeholder object called None.
- None object commonly used to initialize names and objects.

In [165]:

```
1 > 2, 1 < 2 # Booleans
```

Out[165]:

```
(False, True)
```

In [166]:

```
bool('spam') # Object's Boolean value
```

Out[166]:

```
True
```

In [167]:

```
X = None # None placeholder
print(X)
```

```
None
```

In [168]:

```
L = [None] * 10 # Initialize a list of 10 Nones
L[9] = 'Reshma'
```

In [169]:

```
L
```

Out[169]:

```
[None, None, None, None, None, None, None, None, None, 'Reshma']
```

type object

- it is returned by the type built-in function.
- it is an object that gives the type of another object.

In [2]:

```
L = [1,2,3]
D = {'a':1,'b':2,'c':3}
S = {1,2,3,4}
T = (1,2,7,9)
St = 'Hitesh'
N = 818
B = True
X = None
type(L),type(D),type(S),type(T),type(St),type(N),type(B),type(X)
```

Out[2]:

```
(list, dict, set, tuple, str, int, bool, NoneType)
```

In [3]:

```
type(type(L))
```

Out[3]:

```
type
```

- most practical application of type object is...it allows code to check the types of the objects it processes.
- there are three ways to check the types of the objects in a Python script.

In [4]:

```
if type(L) == type([]): # Type testing
    print('yes')
```

```
yes
```

In [5]:

```
if type(L) == list: # Using the type name
     print('yes')
```

```
yes
```

In [6]:

```
if isinstance(L, list): # Object-oriented tests
     print('yes')
```

```
yes
```

Classs

In [175]:

```
class Worker:
    def __init__(self, name, pay):
        self.name = name
        self.pay = pay
    def lastName(self):
        return self.name.split()[-1]
    def giveRaise(self, percent):
        self.pay *= (1.0 + percent)
bob = Worker('Bob Smith', 50000)
sue = Worker('Sue Jones', 60000)
```

In [176]:

```
bob.lastName(),sue.lastName()
```

Out[176]:

```
('Smith', 'Jones')
```

In [177]:

```
sue.giveRaise(.10)
```

In [178]:

```
sue.pay
```

Out[178]:

```
66000.0
```

In [ ]:

```

```