California State University Fullerton

CPSC-235P

Python Programming

Stephen T. May

Python Tutorial

Section 5

Data Structures

https://docs.python.org/release/3.9.6/tutorial/index.html

# Slide Notes

- Command typed at the Linux command prompt ($)

```
$ python3.9
```

- Command typed at the Python interpreter command prompt (>>>)

```
>>> Ctrl-D
```

- Python source code

```
print(“Hello world!”)
```

- Mixed example

```
>>> the_world_is_flat = True
>>> if the_world_is_flat:
...     print("Be careful not to fall off!")
...
Be careful not to fall off!
```

# 5.1 More on Lists

- `list.append(x)`
  - Add an item to the end of the list
  - Equivalent to `a[len(a):] = [x]`

- `list.extend(iterable)`
  - Extend the list by appending all the items from the iterable
  - Equivalent to `a[len(a):] = iterable`

- `list.insert(i, x)`
  - Insert an item at a given position
  - The first argument is the index of the element before which to insert, so `a.insert(0, x)` inserts at the front of the list, and `a.insert(len(a), x)` is equivalent to `a.append(x)`

- `list.remove(x)`
  - Remove the first item from the list whose value is equal to x
  - It raises a ValueError if there is no such item

```
>>> fruits = ['orange', 'apple']
>>> fruits
['orange', 'apple']
>>>
>>> fruits.append('pear')
>>> fruits
['orange', 'apple', 'pear']
>>>
>>> fruits.extend(['kiwi', 'grape'])
>>> fruits
['orange', 'apple', 'pear', 'kiwi', 'grape']
>>>
>>> fruits.insert(1,'mango')
>>> fruits
['orange', 'mango', 'apple', 'pear', 'kiwi', 'grape']
>>>
>>> fruits.remove('pear')
>>> fruits
['orange', 'mango', 'apple', 'kiwi', 'grape']
```

# 5.1 More on Lists (cont.)

- `list.pop([i])`
  - Remove the item at the given position in the list, and return it
  - If no index is specified, `a.pop()` removes and returns the last item in the list
  - The square brackets around the i in the method signature denote that the parameter is optional, not that you should type square brackets at that position. You will see this notation frequently in the Python Library Reference

- `list.clear()`
  - Remove all items from the list
  - Equivalent to `del a[:]`

- `list.index(x[, start[, end]])`
  - Return zero-based index in the list of the first item whose value is equal to x
  - Raises a ValueError if there is no such item
  - The optional arguments start and end are interpreted as in the slice notation and are used to limit the search to a particular subsequence of the list
  - The returned index is computed relative to the beginning of the full sequence rather than the start argument

```
>>> fruits
['orange', 'mango', 'apple', 'kiwi', 'grape']
>>> x = fruits.pop(3)
>>> fruits
['orange', 'mango', 'apple', 'grape']
>>> x
'kiwi'
>>>
>>> fruits.clear()
>>> fruits
[]
>>> fruits = ['pear', 'grape', 'pear', 'apple', 'pear', 'kiwi']
>>> fruits.index('pear', 1, 5)
2
```

# 5.1 More on Lists (cont.)

- `list.count(x)`
  - Return the number of times x appears in the list

- `list.sort(*, key=None, reverse=False)`
  - Sort the items of the list in place
  - the arguments can be used for sort customization
  - Mixed datatypes will not sort
  - see `sorted()` for their explanation

- `list.reverse()`
  - Reverse the elements of the list in place

- `list.copy()`
  - Return a shallow copy of the list
  - Equivalent to `a[:]`

```
>>> fruits = ['pear', 'grape', 'pear', 'apple', 'pear', 'kiwi']
>>> fruits.count('pear')
3
>>>
>>> fruits = ['pear', 'grape', 'Pear', 'apple', 'pear', 'kiwi']
>>> fruits.sort()
>>> fruits
['Pear', 'apple', 'grape', 'kiwi', 'pear', 'pear']
>>>
>>> fruits = ['pear', 'grape', 'Pear', 'apple', 'pear', 'kiwi']
>>> fruits.sort(key=str.lower)
>>> fruits
['apple', 'grape', 'kiwi', 'pear', 'Pear', 'pear']
>>>
>>> fruits = ['pear', 'grape', 'Pear', 'apple', 'pear', 'kiwi']
>>> fruits.sort(reverse=True)
>>> fruits
['pear', 'pear', 'kiwi', 'grape', 'apple', 'Pear']
>>>
>>> fruits = ['pear', 'grape', 'Pear', 'apple', 'pear', 'kiwi']
>>> fruits.reverse()
>>> fruits
['kiwi', 'pear', 'apple', 'Pear', 'grape', 'pear']
>>>
>>> fruits.copy()
['kiwi', 'pear', 'apple', 'Pear', 'grape', 'pear']
```

# 5.1.1 Using Lists as Stacks

- The list methods make it very easy to use a list as a stack
- Last element added is the first element retrieved ("last-in, first-out")
- To add an item to the top of the stack, use `append()`
- To retrieve an item from the top of the stack, use `pop()` without an explicit index

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

# 5.1.2 Using Lists as Queues

- It is also possible to use a list as a *queue*
- The first element added is the first element retrieved ("first-in, first-out")
- However, lists are not efficient for this purpose
- While appends and pops from the end of list are fast, doing inserts or pops from the beginning of a list is slow (because all of the other elements have to be shifted by one)
- To implement a queue, use `collections.deque` which was designed to have fast appends and pops from both ends

```
>>> from collections import deque
>>> queue = deque(["Eric", "John", "Michael"])
>>> queue.append("Terry")           # Terry arrives
>>> queue.append("Graham")          # Graham arrives
>>> queue.popleft()                 # The first to arrive now
leaves
'Eric'
>>> queue.popleft()                 # The second to arrive now
leaves
'John'
>>> queue                           # Remaining queue in order of
arrival
deque(['Michael', 'Terry', 'Graham'])
```

# 5.1.3 List Comprehensions

- *List comprehensions* provide a concise way to create lists
- Common applications are to make new lists where each element is the result of some operations applied to each member of another *sequence* or *iterable*, or to create a subsequence of those elements that satisfy a certain condition
- A list comprehension consists of brackets containing an expression followed by a for clause, then zero or more for or if clauses
- The result will be a new list resulting from evaluating the expression in the context of the for and if clauses which follow it

```
>>> squares = []
>>> for x in range(10):
...     squares.append(x**2)
...
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>>
>>> squares = list(map(lambda x: x**2, range(10)))
>>>
>>> squares = [x**2 for x in range(10)] # list comprehension
```

```
>>> combs = []
>>> for x in [1,2,3]:
...     for y in [3,1,4]:
...         if x != y:
...             combs.append((x, y))
...
>>> combs
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
>>>
>>> # list comprehension
>>> [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

# 5.1.3 List Comprehensions (cont.)

```
>>> vec = [-4, -2, 0, 2, 4]
>>> # create a new list with the values doubled
>>> [x*2 for x in vec]
[-8, -4, 0, 4, 8]
>>> # filter the list to exclude negative numbers
>>> [x for x in vec if x >= 0]
[0, 2, 4]
>>> # apply a function to all the elements
>>> [abs(x) for x in vec]
[4, 2, 0, 2, 4]
>>> # call a method on each element
>>> freshfruit = ['  banana', '  loganberry ', 'passion fruit  ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
>>> # create a list of 2-tuples like (number, square)
>>> [(x, x**2) for x in range(6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
>>> # the tuple must be parenthesized, otherwise an error is raised
>>> [x, x**2 for x in range(6)]
  File "<stdin>", line 1, in <module>
    [x, x**2 for x in range(6)]
               ^
SyntaxError: invalid syntax
>>> # flatten a list using a listcomp with two 'for'
>>> vec = [[1,2,3], [4,5,6], [7,8,9]]
>>> [num for elem in vec for num in elem]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

# 5.1.4 Nested List Comprehensions

- The initial expression in a list comprehension can be any arbitrary expression, including another list comprehension

```
>>> matrix = [
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12],
... ]
```

```
>>> [[row[i] for row in matrix] for i in range(4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

```
>>> transposed = []
>>> for i in range(4):
...     transposed.append([row[i] for row in matrix])
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

- In the real world, you should prefer built-in functions to complex flow statements. The `zip()` function would do a great job for this use case

```
>>> transposed = []
>>> for i in range(4):
...     # the following 3 lines implement the nested listcomp
...     transposed_row = []
...     for row in matrix:
...         transposed_row.append(row[i])
...     transposed.append(transposed_row)
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

```
>>> list(zip(*matrix))
[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]
```

# 5.2 The `del` Statement

- There is a way to remove an item from a list given its index instead of its value: the `del` statement
- This differs from the `pop()` method which returns a value
- The `del` statement can also be used to remove slices from a list or clear the entire list (which we did earlier by assignment of an empty list to the slice)
- `del` can also be used to delete entire variables
- Referencing the variable name hereafter is an error

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
>>> del a
>>> a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
>>>
```

# 5.3 Tuples and Sequences

- A *tuple* consists of a number of values separated by commas
- On output tuples are always enclosed in parentheses, so that nested tuples are interpreted correctly
- Tuples may be input with or without surrounding parentheses, although often parentheses are necessary anyway (if the tuple is part of a larger expression)
- It is not possible to assign to the individual items of a tuple, however it is possible to create tuples which contain mutable objects, such as lists
- Tuples are immutable, and usually contain a heterogeneous sequence of elements that are accessed via unpacking or indexing
- Note: *Lists* are mutable, and their elements are usually homogeneous and are accessed by iterating over the list

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Tuples may be nested:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
>>> # Tuples are immutable:
... t[0] = 88888
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> # but they can contain mutable objects:
... v = ([1, 2, 3], [3, 2, 1])
>>> v
([1, 2, 3], [3, 2, 1])
```

# 5.3 Tuples and Sequences (cont.)

- A special problem is the construction of tuples containing 0 or 1 items: the syntax has some extra quirks to accommodate these
- Empty tuples are constructed by an empty pair of parentheses
- A tuple with one item is constructed by following a value with a comma (it is not sufficient to enclose a single value in parentheses)
- *Tuple packing* are when values are packed together into a tuple
- *Sequence unpacking* is the reverse and requires that there are many variables on the left side of the equals sign as there are elements in the sequence

```
>>> empty = ()
>>> singleton = 'hello',    # <-- note trailing comma
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('hello',)
```

```
>>> t = 12345, 54321, 'hello!'
>>> t
(12345, 54321, 'hello!')
>>>
>>> x, y, z = t
>>> x
12345
>>> y
54321
>>> z
'hello!'
>>> t
(12345, 54321, 'hello!')
```

# 5.4 Sets

- Python also includes a data type for *sets*
- A set is an unordered collection with no duplicate elements
- Basic uses include membership testing and eliminating duplicate entries
- Set objects also support mathematical operations like union, intersection, difference, and symmetric difference
- Curly braces or the `set()` function can be used to create sets
- Note: to create an empty set you have to use `set()`, not `{}`; the latter creates an empty dictionary, a data structure that we discuss in the next section
- *Set comprehensions* are also supported

```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange',
'banana'}
>>> print(basket)                      # show that duplicates have
been removed
{'orange', 'banana', 'pear', 'apple'}
>>> 'orange' in basket                 # fast membership testing
True
>>> 'crabgrass' in basket
False

>>> # Demonstrate set operations on unique letters from two words
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                  # unique letters in a
{'a', 'r', 'b', 'c', 'd'}
>>> a - b                              # letters in a but not in b
{'r', 'd', 'b'}
>>> a | b                              # letters in a or b or both
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b                              # letters in both a and b
{'a', 'c'}
>>> a ^ b                              # letters in a or b but not
both
{'r', 'd', 'b', 'm', 'z', 'l'}
```

```
>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
{'r', 'd'}
```

# 5.5 Dictionaries

- It is best to think of a *dictionary* as a set of *key: value* pairs, with the requirement that the keys are unique (within one dictionary)
- Placing a comma-separated list of key:value pairs within the braces adds initial key:value pairs to the dictionary; this is also the way dictionaries are written on output
- A pair of braces creates an empty dictionary: `{}`
- Unlike *sequences*, which are indexed by a range of numbers, dictionaries are indexed by keys, which can be any immutable type; strings and numbers can always be keys
- *Tuples* can be used as keys if they contain only strings, numbers, or tuples; if a tuple contains any mutable object either directly or indirectly, it cannot be used as a key
- You can't use *lists* as keys, since lists can be modified in place using index assignments, slice assignments, or methods like `append()` and `extend()`

```
>>> banking = {4725: 'checking', 5207: 'savings', 2786: 'loan'}
>>> banking
{4725: 'checking', 5207: 'savings', 2786: 'loan'}
>>>
>>> stocks = {}
>>> stocks
{}
```

# 5.5 Dictionaries (cont.)

- The main operations on a dictionary are storing a value with some key and extracting the value given the key
- It is also possible to delete a key:value pair with `del`
- If you store using a key that is already in use, the old value associated with that key is forgotten
- It is an error to extract a value using a non-existent key
- Performing `list(d)` on a dictionary returns a list of all the keys used in the dictionary, in insertion order (if you want it sorted, just use `sorted(d)` instead)
- To check whether a single key is in the dictionary, use the `in` keyword

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'jack': 4098, 'sape': 4139, 'guido': 4127}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'jack': 4098, 'guido': 4127, 'irv': 4127}
>>> list(tel)
['jack', 'guido', 'irv']
>>> sorted(tel)
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
>>> 'jack' not in tel
False
```

# 5.5 Dictionaries (cont.)

- The `dict()` constructor builds dictionaries directly from sequences of key-value pairs

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

- Dictionary comprehensions can be used to create dictionaries from arbitrary key and value expressions

```
>>> {x: x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}
```

- When the keys are simple strings, it is sometimes easier to specify pairs using keyword arguments

```
>>> dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

# 5.6 Looping Techniques

- When looping through *dictionaries*, the key and corresponding value can be retrieved at the same time using the `items()` method

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.items():
...     print(k, v)
...
gallahad the pure
robin the brave
```

- When looping through a *sequence*, the position index and corresponding value can be retrieved at the same time using the `enumerate()` function

```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...     print(i, v)
...
0 tic
1 tac
2 toe
```

- To loop over two or more *sequences* at the same time, the entries can be paired with the `zip()` function

```
>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue']
>>> for q, a in zip(questions, answers):
...     print('What is your {0}?  It is {1}.'.format(q, a))
...
What is your name?  It is lancelot.
What is your quest?  It is the holy grail.
What is your favorite color?  It is blue.
```

# 5.6 Looping Techniques (cont.)

- To loop over a *sequence* in reverse, first specify the sequence in a forward direction and then call the `reversed()` function

```
>>> for i in reversed(range(1, 10, 2)):
...     print(i)
...
9
7
5
3
1
```

- To loop over a *sequence* in sorted order, use the `sorted()` function which returns a new sorted list while leaving the source unaltered

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange',
'banana']
>>> for i in sorted(basket):
...     print(i)
...
apple
apple
banana
orange
orange
pear
```

# 5.6 Looping Techniques (cont.)

- Using `set()` on a *sequence* eliminates duplicate elements. The use of `sorted()` in combination with `set()` over a sequence is an idiomatic way to loop over unique elements of the sequence in sorted order

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange',
'banana']
>>> for f in sorted(set(basket)):
...     print(f)
...
apple
banana
orange
pear
```

- It is sometimes tempting to change a *list* while you are looping over it; however, it is often simpler and safer to create a new list instead

```
>>> import math
>>> raw_data = [56.2, float('NaN'), 51.7, 55.3, 52.5, float('NaN'),
47.8]
>>> filtered_data = []
>>> for value in raw_data:
...     if not math.isnan(value):
...         filtered_data.append(value)
...
>>> filtered_data
[56.2, 51.7, 55.3, 52.5, 47.8]
```

# 5.7 More on Conditions

- The conditions used in *while* and *if* statements can contain any operators, not just comparisons
- The comparison operators *in* and *not in* check whether a value occurs (does not occur) in a sequence
- The operators *is* and *is not* compare whether two objects are really the same object
- All comparison operators have the same priority, which is lower than that of all numerical operators
- Comparisons can be chained. For example, *a < b == c* tests whether *a* is less than *b* and moreover *b* equals *c*

```
>>> if 'apple' in ['grape', 'apple', 'orange']:
...     print('TRUE')
... else:
...     print('FALSE')
...
TRUE
>>> x = 'apple'
>>> y = ['apple']
>>> if x is y:
...     print('TRUE')
... else:
...     print('FALSE')
...
FALSE
>>> a = 1
>>> b = 2
>>> c = 3
>>> if a < b == c:
...     print('TRUE')
... else:
...     print('FALSE')
...
FALSE
>>> if a < b and b == c:
...     print('TRUE')
... else:
...     print('FALSE')
...
FALSE
```

# 5.7 More on Conditions (cont.)

- Comparisons may be combined using the Boolean operators *and* and *or*, and the outcome of a comparison (or of any other Boolean expression) may be negated with not
- These have lower priorities than comparison operators
- Between them, *not* has the highest priority and *or* the lowest
  - A and not B or C ⇔ (A and (not B)) or C
- Parentheses can be used to express the desired composition

```
>>> a = True
>>> b = False
>>> if a and b:
...     print('TRUE')
... else:
...     print('FALSE')
...
FALSE
>>> if not(a and b):
...     print('TRUE')
... else:
...     print('FALSE')
...
TRUE
>>> a = True
>>> b = False
>>> c = False
>>> if a or not b and c:
...     print('TRUE')
... else:
...     print('FALSE')
...
TRUE
```

# 5.7 More on Conditions (cont.)

- The Boolean operators *and* and *or* are so-called *short-circuit* operators: their arguments are evaluated from left to right, and evaluation stops as soon as the outcome is determined
  - if A and C are true but B is false, A and B and C does not evaluate the expression C
- When used as a general value and not as a Boolean, the return value of a short-circuit operator is the last evaluated argument
- It is possible to assign the result of a comparison or other Boolean expression to a variable
- Note that in Python, unlike C, assignment inside expressions must be done explicitly with the *walrus operator* `:=`
- This avoids a common class of problems encountered in C programs: typing `=` in an expression when `==` was intended

```
>>> a = True
>>> b = False
>>> c = True
>>> if a and b and c:
...     print('TRUE')
... else:
...     print('FALSE')
...
FALSE
>>>
>>> d = a and b and c
>>> d
False
>>>
>>> x = 1
>>> x
1
>>> if (x := x + 1) == 2:
...     print('TRUE')
... else:
...     print('FALSE')
...
TRUE
>>> x
2
```

# 5.8 Comparing Sequences and Other Types

- Sequence objects typically may be compared to other objects with the same sequence type
- The comparison uses *lexicographical* ordering
  - first the first two items are compared,
  - if they differ this determines the outcome of the comparison
  - if they are equal, the next two items are compared
  - and so on, until either sequence is exhausted
- If two items to be compared are themselves sequences of the same type, the lexicographical comparison is carried out recursively
- If all items of two sequences compare equal, the sequences are considered equal
- If one sequence is an initial sub-sequence of the other, the shorter sequence is the smaller (lesser) one
- Lexicographical ordering for strings uses the *Unicode* code point number to order individual characters

```
>>> (1, 2, 3) ==  (1, 2, 4)
False
>>> (1, 2, 3) <  (1, 2, 4)
True
>>> (1, 2, 3) >  (1, 2, 4)
False
>>> [1, 2, 3] < [1, 2, 4]
True
>>> 'ABC' < 'C' < 'Pascal' < 'Python'
True
>>> (1, 2, 3, 4) < (1, 2, 4)
True
>>> (1, 2, 4) < (1, 2, 3, 4)
False
>>> (1, 2) < (1, 2, -1)
True
>>> (1, 2) < (1, 2)
False
>>> (1, 2, 3) == (1.0, 2.0, 3.0)
True
>>> (1, 2, ('aa', 'ab')) < (1, 2, ('abc', 'a'), 4)
True
>>> ('aa', 'ab') < ('abc', 'a')
True
```