Applied Computing with Python

SAT 4650

Lecture 3

Dr. Neerav Kaushal

Department of Applied Computing

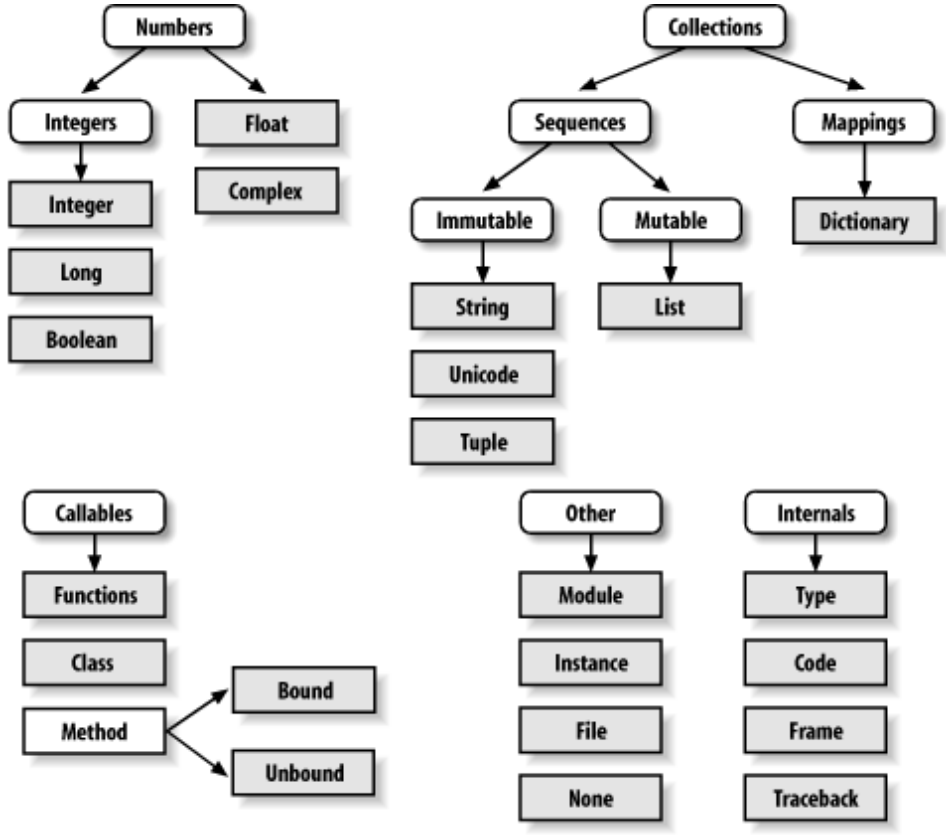
Outline

- Sequences
- Dictionaries

Sequences

Everything is an object

- Integers (default for numbers): z = 5
- Floats: x = 3.456
- Strings:
  - Can use “…” or ‘…’ to specify. "foo" == 'foo’
  - Unmatched can occur within the string: “John’s” or ‘John said “fool”.’
  - Use triple double-quotes for multi-line strings or strings that contain both ‘ and “ inside of them:
    - “““a‘b“c”””
    - print(“””a'b"c”””) -> a’b”c
  - * is repeat, the number is how many times
    - new_str = ’spam-spam-’
    - new_str*3 = ‘spam-spam-spam-spam-spam-spam-’
    - print("ab"*3)→ababab




```
>>> a=12
>>> type(a)
<class 'int'>
>>> a=12.0
>>> type(a)
<class 'float'>
>>> a='12'
>>> type(a)
<class 'str'>
```

```
>>> l1=(1,2,3)
>>> len(l1)
3
>>> type(l1)
<class 'tuple'>
>>>
```

Michigan Tech

object.something() notation

- An object has methods and properties
- Methods and properties of an object can be called by dot notation
- In practice, dot notation looks like: object.method(…)
- It means that the object in front of the dot is calling a method that is associated with that object's type
- Example:
  - my_str = 'Python Rules!’
  - new_string = my_str.upper()
  - new_string = 'PYTHON RULES!'

Types of python sequences

- Sequences are containers that hold objects
- Finite, ordered, indexed by integers
- List is a mutable ordered sequence of items of mixed types:
  - [“one”, “two”, 3]
- Tuple is an immutable ordered sequence of items, and the items can be of mixed types, including collection types:
  - (1, “a”, [100], “foo”)
- Strings are an immutable ordered sequence of chars and are conceptually very much like a tuple:
  - “foo bar”
- All three sequence types (tuples, strings, and lists) share much of the same syntax and functionality

- Key differences:
  - Tuples and strings are immutable
  - Lists are mutable
  - The operations shown in this section can be applied to all sequence types
  - Most examples will just show the operation performed on one

- Define tuples using parentheses and commas

```
>>> tup = (23, 'abc', 4.56, (2,3), 'def')
```

- Define lists using square brackets and commas

```
>>> lis = ["abc", 34, 4.34, 23]
```

- Define strings using quotes (“, ‘, or “””)

```
>>> st = "Hello World"
>>> st = 'Hello World'
>>> st = """This is a multi-line
string that uses triple quotes."""
```

Michigan Tech

Accessing sequence elements - Indexing

- Access individual members of a tuple, list, or string using square bracket "array" notation
- *Note that all are 0 based…*

- Define tuples using parentheses and commas **tup[1]**

```
>>> tup = (23, 'abc', 4.56, (2,3), 'def')
```

- Define lists using square brackets and commas

```
>>> lis = ["abc", 34, 4.34, 23]
```
lis[1]

- Define strings using quotes (", ', or """)

```
>>> st = "Hello World"
>>> st = 'Hello World'
>>> st = """This is a multi-line
string that uses triple quotes."""
```
st[1]

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

- Positive index: count from the left, starting with 0

```
>>> t[1]
```
'abc'

- Negative index: count from right, starting with –1

```
>>> t[-3]
```
4.56

Michigan Tech

Accessing sequence elements - Slicing

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

- Returns copy of container with subset of original members. Start copying at first index, and stop copying before the second index

```
>>> t[1:4]('abc', 4.56, (2,3))
```

- You can also use negative indices

```
>>> t[1:-1]('abc', 4.56, (2,3))
```

- Omit first index to make a copy starting from the beginning of container

```
>>> t[:2](23, 'abc')
```

- Omit second index to make a copy starting at 1st index and going to end of the container

```
>>> t[2:](4.56, (2,3), 'def')
```

- [:] makes a copy of an entire sequence

```
>>> t[:]
(23, 'abc', 4.56, (2,3), 'def')
```

- Note the difference between these two lines for mutable sequences

```
>>> l2 = l1
# both refer to same ref,
# changing one affects both

>>> l2 = l1[:]
# independent copies
# two references
```

show in code with element change

Michigan Tech

The in and + operators

- Boolean test whether a value is inside a container

```
>>> t = [1, 2, 4, 5]
>>> 3 in t
False
>>> 4 in t
True
>>> 4 not in t
False
```

- For strings, tests for substrings

```
>>> a = 'abcde'
>>> 'c' in a
True
>>> 'cd' in a
True
>>> 'ac' in a
False
```

- The + operator produces a new tuple, list, or string whose value is the concatenation of its arguments

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

>>> [1, 2, 3] + [4, 5, 6]
    [1, 2, 3, 4, 5, 6]

>>> "Hello" + " " + "World"
    "Hello World"
```

Mutability

- Lists are mutable

```
>>> t = [1, 2, 4, 5]
>>> t[1] = 99
     [1, 99, 4, 5]
```

- We can change lists in place
- Name t still points to the same memory reference after execution

- Tuples are immutable: you can't change them

```
>>> t = (1, 2, 4, 5)
>>> t[1] = 99
     [Traceback (most recent call last):
   File "<pyshell#75>", line 1, in -toplevel-
     t[1] = 99
 TypeError: object doesn't support item assignment
```

- This immutability means that they are faster to work with than lists

Functions vs. methods

- Some operations are functions and other operations are methods
  - Remember that (almost) everything is an object
  - You should just learn (and remember or lookup) which operations are functions and which are methods
- Methods are associated with the objects of the class they belong to. Functions are not associated with any object. We can invoke a function just by its name. Functions operate on the data you pass to them as arguments

len() is a function on collections that returns the number of things they contain

```
>>> len(['a', 'b', 'c'])
3
>>> len(('a','b','c'))
3
>>> len("abc")
3
```

index() is a method on collections that returns the index of the 1st occurrence of its argument

```
>>> ['a','b','c'].index('a')
0
>>> ('a','b','c').index('b')
1
>>> "abc".index('c')
2
```

Lists methods

- Lists have many methods, including index, count, append, remove, reverse, sort, etc.
- Many of these modify the list

```
>>> l = [1,3,4]
>>> l.append(0)       # adds a new element to the end of the list
>>> l
[1, 3, 4, 0]
>>> l.insert(1,200)   # insert value just before an index
>>> l
[1, 200, 3, 4, 0]
>>> l.reverse()       # reverse the list in place
>>> l
[0, 4, 3, 200, 1]
>>> l.sort()          # sort the elements.  Optional arguments can give
>>> l                 # the sorting function and direction
[0, 1, 3, 4, 200]
>>> l.remove(3)       # remove first occurence of element from list
>>> l
[0, 1, 4, 200]
```

What is unicode?

- Computers store everything as numbers, but humans use characters:
  - English letters: A, b, z
  - Numbers: 0–9
  - Symbols: @, $, %
  - Other languages: **न**, 中, α
  - Emojis: 😀, 🚀
- Early systems only handled English characters (ASCII). That broke the moment you tried to use:
  - Accents (é, ñ)
  - Non-latin scripts
  - Emojis
- Unicode was created to assign a unique number to every character used in human writing. Unicode assigns each character a unique integer code. U+ means unicode followed by a hexadecimal number
- Therefore, “s = "Hello π 😀" is not a problem for python

Character	Unicode code point
A	U+0041
a	U+0061
0	U+0030
π	U+03C0
中	U+4E2D
😀	U+1F600

```
In [18]: ord('a')
Out[18]: 97

In [19]: ord('A')
Out[19]: 65
```

```
In [15]: ord('π')
Out[15]: 960

In [16]: ord('中')
Out[16]: 20013

In [17]: ord('😀')
Out[17]: 128512
```

String's functions and methods

- ord(ch) --- returns the unicode value of a single-character ch
- chr(num) returns the character represented by the Unicode value num
- Notes: Unicode, developed by the Unicode Consortium, can map characters to numbers in multiple languages. The first 128 character in the unicode system is the same as the ASCII characters.
- ord('a') => 97

  ord('c') => 99

  chr(104) => 'h'

  chr(97+13) => 'n'

- `my_str = 'hello'`
- find searches the string for a specified value and returns the index of first occurence
  - `my_str.find('l')` will return 2
- `myTuple = ("John", "Peter", "Vicky")`
- join joins the elements of an iterable to the end of the string
  - `x = "-".join(myTuple)` will output John-Peter-Vicky
  - `x = "".join(myTuple)` will output JohnPeterVicky
  - `x = " ".join(myTuple)` will output John Peter Vicky
- `txt = "I could eat bananas all day"`
- partition returns a tuple where the string is parted into three parts:
  - `x = txt.partition("bananas")` will return ('I could eat ', 'bananas', ' all day')
- replace() replaces a specified phrase with another specified phrase
  - `x = txt.replace("bananas", "apples")` will output "I like apples"
- split splits a string into a list where each word is a list item
  - `x = txt.split()` will output ['welcome', 'to', 'the', 'jungle']

String’s functions and methods

```
In [20]: ord('a')
Out[20]: 97

In [21]: ord(a)
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[21], line 1
----> 1 ord(a)

NameError: name 'a' is not defined
```

```
In [20]: ord('xx')
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[20], line 1
----> 1 ord('xx')

TypeError: ord() expected a character, but string of length 2 found
```

Tuples in more detail

- The comma is the tuple creation operator, not parentheses

  >>> 1,

  (1,)

- Python shows parentheses for clarity (best practice)

  >>> (1,)

  (1,)

- Don't forget the comma!

  >>> (1)

  1

- Trailing comma only required for singletons. For more than one element, (1,4,-7)
- Empty tuples have a special syntactic form

  >>> ()

  ()

  >>> tuple()

  ()

- Lists are slower but more powerful than tuples
- Lists can be modified and they have many handy operations and methods
- Tuples are immutable & have fewer features
- Sometimes an immutable collection is required (e.g., as a hash key)
- Convert tuples and lists using list() and tuple():
  - mylst = list(mytup)
  - mytup = tuple(mylst)

```
>>> a = [1,2,3,4,5]
>>> type(a)
 <class 'list'>
>>> b = tuple(a)
>>> type(b)
 <class 'tuple'>
>>>
```

```
>>> a = (1,2,3,4,5)
>>> type(a)
 <class 'tuple'>
>>> b = list(a)
>>> type(b)
 <class 'list'>
>>>
```

Another module like math is random

- random module can be used to generate pseudo-random numbers. More details here: Read more here: https://docs.python.org/3/library/random.html
- random.randrange(*n*) creates a random number between 0 and n (not including n)
- random.randrange(m,*n*) creates a random number between m and n (not including n)
- random.choice(seq) Here sequence can be a list, string, tuple) returns a single item from the sequence
- random.choices(seq, k=N) returns a list(length=N) with the randomly selected elements from the specified sequence
- random.shuffle(seq) shuffles a sequence
- random.seed(42) fixes a seed so each time you run the program, it gives you the same random numbers
- For more random functions (like random floats for example, use numpy.random class)

```
In [1]: import random

In [2]: random.randrange(7)
Out[2]: 4

In [3]: random.randrange(3,7)
Out[3]: 5

In [4]: mylist = ['north', 'south', 'east', 'west']

In [5]: random.choice(mylist)
Out[5]: 'east'

In [6]: random.shuffle(mylist)

In [7]: mylist
Out[7]: ['north', 'east', 'west', 'south']

In [8]: random.choices(mylist, k=9)
Out[8]: ['west', 'north', 'north', 'west', 'east', 'south', 'west', 'south', 'east']
```

Dictionary

What is a dictionary? Why another datatype?

- Dictionary (dict) stores a set of key-value pairs (like a “bag” of labeled items)
- Each entry looks like this - key : value
- Real-world analogy: username : password, course : grade, word : count
- Dictionaries use curly braces { }
- Each pair is separated by commas
- Each pair uses a colon :
- Keys are unique (you can’t have the same key twice)
- Dictionaries are unordered (don’t rely on position)

```
In [16]: accounts = {
    ...:     "jason": "QEr%@$#",
    ...:     "tom": "yYT%78",
    ...:     "koh": "Poe@!!234"
    ...: }

In [17]: accounts.keys()
Out[17]: dict_keys(['jason', 'tom', 'koh'])

In [18]: accounts.values()
Out[18]: dict_values(['QEr%@$#', 'yYT%78', 'Poe@!!234'])

In [19]: accounts.items()
Out[19]: dict_items([('jason', 'QEr%@$#'), ('tom', 'yYT%78'), ('koh', 'Poe@!!234')])
```

More on dictionary

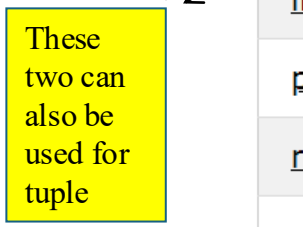
- We know that lists use number indices (listy[0], listy[1], …). Lists: lookup by position
- Dictionaries use keys as “lookup tags” rather than indices. This lets you access values using d[key]. Dictionaries: lookup by key
- Both lists and dicts are mutable (you can change them)
- KeyError happens if you use a key that doesn’t exist
- Use in to check if a key exists
- Add or update a value: d[key] = value
- Delete one item: del(d[key])
- Delete everything: d.clear()
- Wrap with list() to convert to a list. Also called ?
  - type conversion
- Safe lookup with get()
  - d.get(key) → value or None
  - d.get(key, default) → value or default if missing

show code on each utility

Extras

More list methods

Method	Description
append()	Adds an element at the end of the list
clear()	Removes all the elements from the list
copy()	Returns a copy of the list
count()	Returns the number of elements with the specified value
extend()	Add the elements of a list (or any iterable), to the end of the current list
index()	Returns the index of the first element with the specified value
insert()	Adds an element at the specified position
pop()	Removes the element at the specified position
remove()	Removes the first item with the specified value
reverse()	Reverses the order of the list
sort()	Sorts the list



Reference: https://docs.python.org/3/tutorial/datastructures.html

More string methods

capitalize( )	lstrip([*chars*])
center(*width* [, *fillchar*])	partition(*sep*)
count(*sub* [, *start* [, *end*]])	replace(*old*, *new* [, *count*])
decode([*encoding* [, *errors*]])	rfind(*sub* [,*start* [,*end*]])
encode([*encoding* [,*errors*]])	rindex(*sub* [, *start* [, *end*]])
endswith(*suffix* [, *start* [, *end*]])	rjust(*width* [, *fillchar*])
expandtabs([*tabsize*])	rpartition(*sep*)
find(*sub* [, *start* [, *end*]])	rsplit([*sep* [,*maxsplit*]])
index(*sub* [, *start* [, *end*]])	rstrip([*chars*])
isalnum( )	split([*sep* [,*maxsplit*]])
isalpha( )	splitlines([*keepends*])
isdigit( )	startswith(*prefix* [, *start* [, *end*]])
islower( )	strip([*chars*])
isspace( )	swapcase( )
istitle( )	title( )
isupper( )	translate(*table* [, *deletechars*])
join(*seq*)	upper( )
lower( )	zfill(*width*)
ljust(width [, *fillchar*])	

- Reference: https://

More from math module

```
import math
print(math.pi)        # constant in math module
print(math.sin(1.0))# a function in math
help(math.pow)        # help info on pow
```

Command name	Description
`abs(`***value***`)`	absolute value
`ceil(`***value***`)`	rounds up
`cos(`***value***`)`	cosine, in radians
`floor(`***value***`)`	rounds down
`log(`***value***`)`	logarithm, base *e*
`log10(`***value***`)`	logarithm, base 10
`max(`***value1***`, `***value2***`)`	larger of two values
`min(`***value1***`, `***value2***`)`	smaller of two values
`round(`***value***`)`	nearest whole number
`sin(`***value***`)`	sine, in radians
`sqrt(`***value***`)`	square root

Constant	Description
`e`	2.7182818...
`pi`	3.1415926...