Data Types in Python

Basic data types

Numeric data

Boolean data

String data

Numeric data types

Integers – positive or negative whole numbers without a decimal point. 4, -19, 0

Floating point numbers - the decimal point can 'float' to different positions. 123.45 is the same as 1.2345x10^2

Boolean data

Can evaluate to be True or False

Useful for representing binary values – On/Off, Yes/No, True/False, Right/Wrong

String data

String of characters

Characters include alphabets, numbers, punctuations, emojis, symbols, etc

Numeric Literals

Fixed values that represent a constant in Python source code

Integer literals: `10`, `-5`, `0b101` for binary, `0o17` for octal, `0xAF` for hexadecimal

Floating point literals: `3.14`, `-2.5e-3`

Complex literals: `2 + 3j`

String literals

Single-line Strings: `'hello'`, `"Python"`

Multiline Strings:

```
'''This is a
 multiline string.'''
```

Raw String Literals: `r'C:\Users\Name'  to treat backslashes as literal characters`

F-String Literals (Formatted String Literals): for embedding expressions `f'The value is {x}'`

Boolean literals

- `True`
- `False`

Special literal

- `None`: Represents the absence of a value.

Data type related functions

- `type()`: Returns the type of a variable.
- `isinstance()`: Checks if a variable belongs to a certain type.

`These functions are for converting from one type to another`

- `int()`: Converts a value to an integer.
- `float()`: Converts a value to a floating-point number.
- `str()`: Converts a value to a string.
- `bool()`: Converts a value to a boolean (True or False).
- `list()`: Converts an iterable (like a string, tuple, or set) to a list.
- `tuple()`: Converts an iterable to a tuple.
- `set()`: Converts an iterable to a set (an unordered collection of unique elements).
- `dict()`: Converts a sequence of key-value pairs (e.g., a list of tuples) to a dictionary.

Collection data types

- Lists
- Tuples
- Generators
- Dictionaries
- Sets
- Arrays, Data series, data frames
- Strings
- Files

Collections

Collections are prepackaged data structures consisting of related data items.Examples of collections:

- Favorite songs on your smartphone
- Contacts list
- A library’s books
- Cards in a card game
- Favorite sports team’s players
- Stocks in an investment portfolio
- Patients in a cancer study
- Shopping list.

Lists

Lists

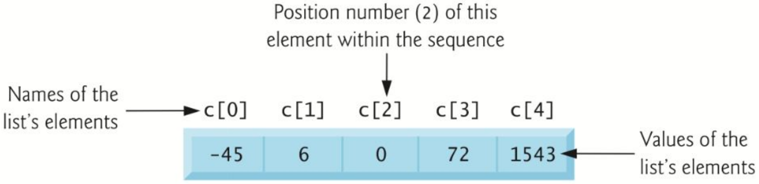
- Sequence; ordered collection of objects
- Lists typically store homogeneous data
- `c = [-45, 6, 0, 72, 1543]`
- but may store heterogeneous data
- `myList = [123, 'spam', 1.23, [89, 90]]`
- `letters1 = list('spam') #unpack the string into a list`
- `l2 = [[1,2,3], [4,5,6], [7,8,9]] #list containing lists`
- `a_list = [] #empty list`
-

Creating a list

```
c1 = []  #create an empty list
c2 = list() #create an empty list
c3 = [-45, 6, 0, 72, 1543]
letters1 = list('spam') #unpack the string into a list
```

Access an item in the list

- Reference a list element by writing the list's name followed by the element's index enclosed in [] (the subscription operator).
- `c = [-45, 6, 0, 72, 1543]`
- `c[0]`
- `c[4]`
- `c[-1] # same as last item`
- `c[-5]`
- `number1, number2, number3 = [2, 3, 5] #unpacks to list items`


Position number (2) of this element within the sequence
Names of the list's elements
c[0]
c[1]
c[2]
c[3]
c[4]
-45
6
0
72
1543
Values of the list's elements

List properties

- Use len(listName) to find size of a list
- Index values must be in range.
- Indices Must Be Integers or Integer Expressions
- `i=1; c[i]`
- Lists Are Mutable
- `c[1] = 100 #assigns 100 to second element in list c`
-

List operations: Adding to a list

the result is a list

- `c = [-45, 6, 0, 72, 1543]`
- `c += [10]`
- `letters1+['a','b'] # appends two items to letters1`
- `letters1 + list("Python") #unpacks 'Python' and appends to letters1`
- `letters1*2 # appends a copy of letters1 to letters1`
- `letters1.append('a1') #appends 'a1' to letters`
- `letters1.insert(0,123) #insert 123 at 0`
- `c + c`
- `c.extend(['xx', 'yy']) #extends the list by two items`

List operations: Removing from a list

- `c = [-45, 6, 0, 72, 1543]`
- `c.pop(1)  #removes the item at index 1; list is shorter now`
- `letters1.remove('y') #finds and removes first occurrence of 'y'`
- `del c[0]  #deletes the item at 0 position`
- `c = [-45, 6, 0, 72, 1543]`
- `del c[0:2] #Removes items 0 and 1`
- `c = [-45, 6, 0, 72, 1543]`
- `c.remove(72) #removes the item and resizes the list`
- `c.clear() #empties the list`

List operations: Comparing lists

```
a = [1, 2, 3]

b = [1, 2, 3]

a == b  #compare for equality

c=[1,2,3,4]

a<c #compare for inequality
```

List operations: Sorting a list

```
c = [45, 26, 10, 72, 43]

c.sort() #modified the list

c.sort(reverse=True) #sorts in descending order

c = [45, 26, 10, 72, 43]

sorted(c) #creates a sorted c; original c is unaffected
```

List operations: Searching Sequences

- `numbers = [3, 7, 1, 4, 2, 8, 5, 6, 3, 5, 6, 7]`
- `numbers.index(5) # index of first element that matches 5`
- `numbers.index(5, 4) #find index of 5 starting from position 4`
- `numbers.index(7, 0, 4) #find position of 7 between 0 and 4`
- `8 in numbers #using the in operator, returns boolean`
- `8 not in numbers`
- `numbers.count(3) # returns number of 3s in the list`

More list operations

reverse, copy, clear

Use

```
help(list) # at the iPython prompt
```

Coding application

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

key = 1000
if key in numbers:

    print(f'found {key} at index {numbers.index(search_key)}')

else:

    print(f'{key} not found')
```

Tuples

Tuples

- Tuples are sequences like lists, but are immutable, - they cannot be changed. They are used to represent fixed collections of items. They are coded in parentheses.
- `t1 = (1,2,3,4)`
- `len(t1) # find how long a tuple is`
- `t1 + (5,6)  # can append`
- `t1[0] = 2 # can't do this; immutable`

Creating a tuple

- student_tuple = ()
- student_tuple = ('Mary', 'Red', 3.3)
- another_student_tuple = 'John', 'Green', 3.3 #Parentheses are optional when creating a tuple.
- a_singleton_tuple = ('red',) #A comma is required to create a one-element tuple.
- student_tuple2 = ('Amanda', 'Blue', [98, 75, 87]) #Tuples May Contain Mutable Objects
- student_tuple2[2][1] = 85

Accessing the tuple elements

- `student_tuple[0]  # use indexes to slice`
- `student_tuple[0] = 2 # can't do this; immutable`
- `t1 = (1,2,3,4)`
- `t1.index(4) # find an item`
- `t1.count(4) # count frequencies`
- `student_tuple = ('Amanda', [98, 85, 87])`
- `first_name, grades = student_tuple #Unpacking Sequences`

Tuple operations: Adding to a tuple

```
tuple1 = (10, 20, 30)

tuple1 += (40, 50)
```

List of tuples

Accessing Indices and Values

Safely with Built-in Function enumerate

Receives an iterable and creates an iterator that, for each element, returns a tuple containing the element’s index and value.

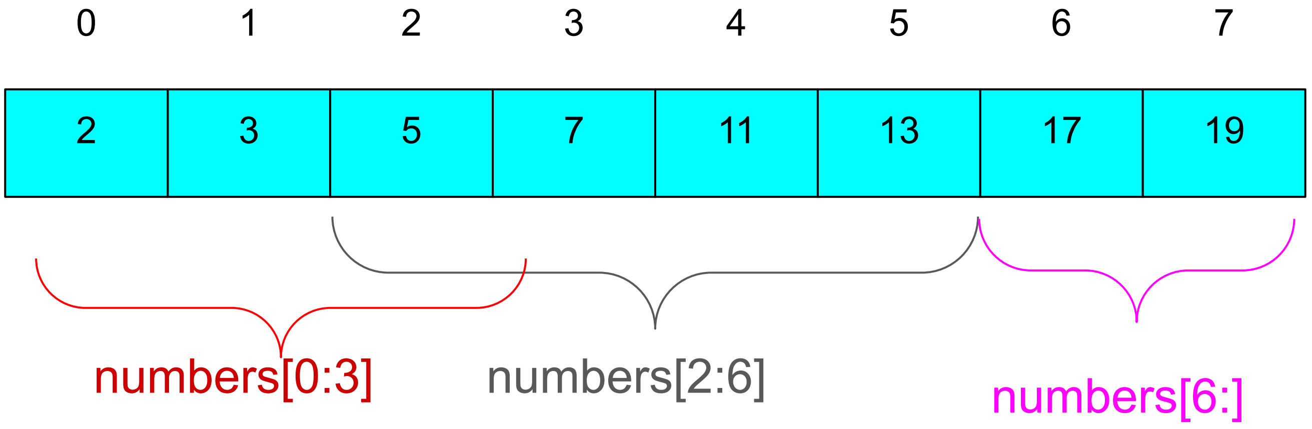
```
colors = ['red', 'orange', 'yellow']

list(enumerate(colors)) #creates a list of tuples
```

Bar chart

```python
"""Displaying a bar chart"""

numbers = [19, 3, 15, 7, 11]

print('\nCreating a bar chart from numbers:')

print(f'Index{"Value":>8}   Bar')

for index, value in enumerate(numbers):

        print(f'{index:>5}{value:>8}   {"*" * value}')
```

Slicing a list

Slice



0	1	2	3	4	5	6	7
2	3	5	7	11	13	17	19

numbers[0:3]

numbers[2:6]

numbers[6:]

Slicing

```
numbers = [2, 3, 5, 7, 11, 13, 17, 19]

numbers[2:6] # get entries from item 2 to item 6, excluding 6

numbers[:6] # Starting from 0, go up to item 6

numbers[6:] # starting from 6, all items after that

numbers[::2] #Get every second item

numbers[::-1] #starting from 0 go -1 index at a time

numbers[0:3] = ['two', 'three', 'five'] #replace the first
```

Using lists

Passing Lists to Functions

```
def modify_elements(items):

    """"Multiplies all element values in items by 2."""

    for i in range(len(items)):

        items[i] *= 2
numbers = [10, 3, 7, 1, 9]
modify_elements(numbers)
numbers
```

Two-dimensional lists

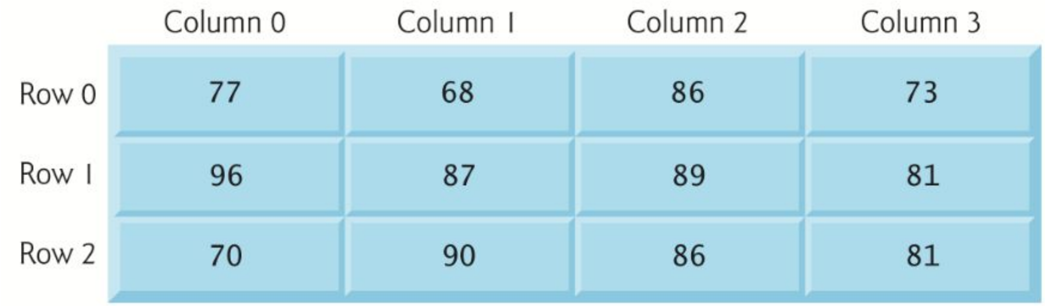
List of lists

- Lists can contain other lists as elements.
- Typical use is to represent tables of values consisting of information arranged in rows and columns.
- To identify a particular table element, we specify two indices—the first identifies the element’s row, the second the element’s column.

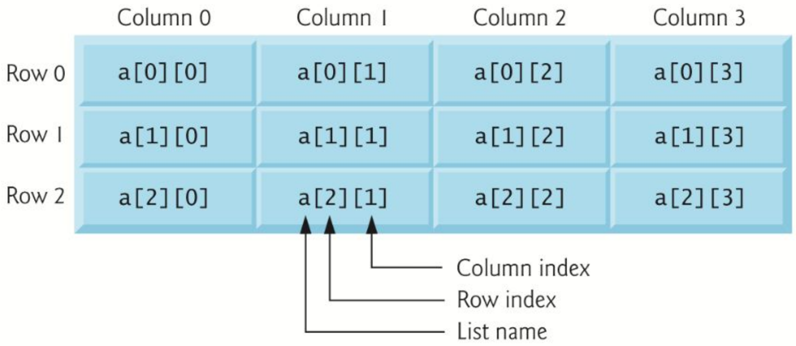
-
  ```
  a = [[77, 68, 86, 73], [96, 87, 89, 81], [70, 90, 86, 81]]
  ```
-

Accessing elements of a list

- `a = [[77, 68, 86, 73], [96, 87, 89, 81], [70, 90, 86, 81]]`



	Column 0	Column 1	Column 2	Column 3
Row 0	77	68	86	73
Row 1	96	87	89	81
Row 2	70	90	86	81



	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Iterating through the elements

```
for i, row in enumerate(matrix1):
    for j, item in enumerate(row):
        print(f'matrix1[{i}][{j}]={item} ', end=' ')
    print()
```

```
a[0][0]=77  a[0][1]=68  a[0][2]=86  a[0][3]=73

a[1][0]=96  a[1][1]=87  a[1][2]=89  a[1][3]=81

a[2][0]=70  a[2][1]=90  a[2][2]=86  a[2][3]=81
```

- Outer for statement iterates over the list’s ows one row at a time.
- During each iteration of the outer for statement, the inner for statement iterates over *each* column in the current row.

Dictionaries

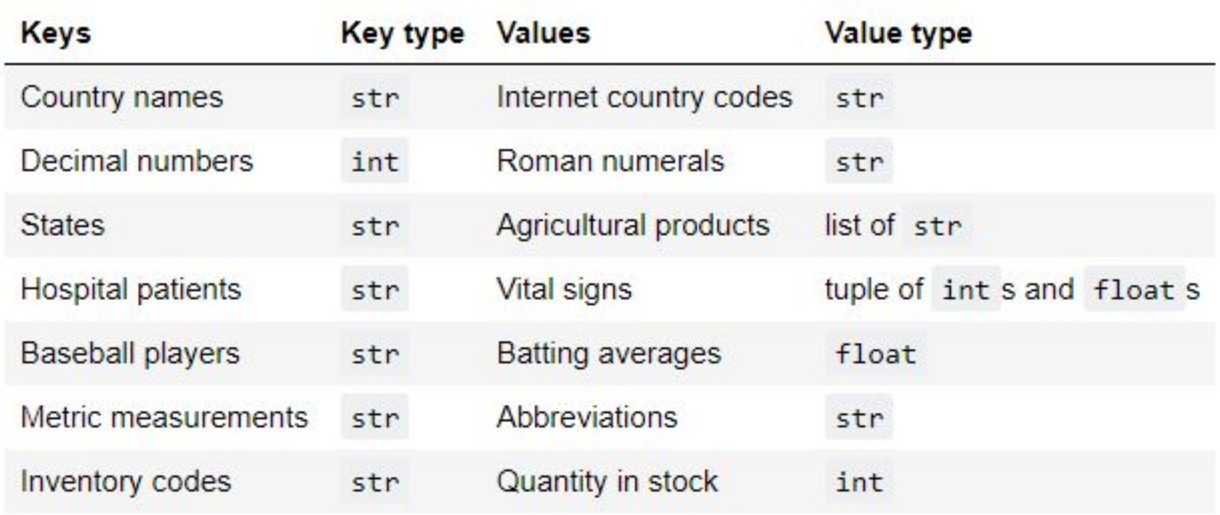
Dictionary

Regular dictionary has *Terms* and *definitions*

In programming, we use *Keys* and *Values*

- A **dictionary** is an *unordered* collection which stores **key–value pairs** that map immutable keys to values, just as a conventional dictionary maps words to definitions.
- fetched by *key* (instead of position as in list). You create dictionaries with literals and access items by key. uses Key-Value pairs
-

Sample dictionary entries



Keys	Key type	Values	Value type
Country names	`str`	Internet country codes	`str`
Decimal numbers	`int`	Roman numerals	`str`
States	`str`	Agricultural products	list of `str`
Hospital patients	`str`	Vital signs	tuple of `int` s and `float` s
Baseball players	`str`	Batting averages	`float`
Metric measurements	`str`	Abbreviations	`str`
Inventory codes	`str`	Quantity in stock	`int`

- Keys must be *immutable* and *unique*.

Creating a Dictionary

Create a dictionary by enclosing in curly braces, {}, a comma-separated list of key–value pairs, each of the form *key: value*.

Create an empty dictionary with {}.

```
country_codes = {'Finland': 'fi', 'South Africa': 'za', 'Nepal': 'np'}

country_codes
country_code.keys()
country_codes.values()
len(country_codes) # Determining if a Dictionary Is Empty

if country_codes: # Determining if a Dictionary Is Empty
    print('country_codes is not empty')
else:
    print('country_codes is empty')
```

Iterating through dictionary items

There are many ways of iterating through a dictionary

```
for k in country_codes : print(country_codes[k])

for item in country_codes.items(): print(item)

for key, value in country_codes.items():
    print (f'Key: {key} Value = {value}')

for t in country_codes.items():
    print(t[0], t[1])
```

Getting the data in and out of a dictionary

```python
days_per_month = {'January': 31, 'February': 28, 'March': 31}
for month, days in days_per_month.items():
    print(f'{month} has {days} days')
```

```python
#Accessing the Value Associated with a Key;  Use square brackets
days_per_month['January']
```

```python
#Updating the Value of an Existing Key–Value Pair
days_per_month['February'] = 29
```

```python
#Adding a New Key–Value Pair

days_per_month['April'] = 30
```

```python
#Removing a Key–Value Pair

del days_per_month['April']
```

```python
#Remove an entry and show what it is using pop

days_per_month.pop('January')
```

```python
#Testing Whether a Dictionary Contains a Specified Key

'January' in days_per_month
```

Dictionary Methods keys and values

```
months = {'January': 1, 'February': 2, 'March': 3}

print (list(months.keys()))

print (list(months.values()))

print (list(months.items()))

for month_name in sorted(months.keys()):  # to process in sorted order

    print(month_name, end='  ')
```

Example: Dictionary of Student Grades

```
# fig06_01.py
"""Using a dictionary to represent an instructor's grade book."""
grade_book = {
    'Susan': [92, 85, 100],
    'Eduardo': [83, 95, 79],
    'Azizi': [91, 89, 82],
    'Pantipa': [97, 91, 92]
}
#compute the class average of totals

sum_of_grades = 0

for student in grade_book.keys():

    print(student, grade_book[student], sum(grade_book[student]))

    sum_of_grades += sum(grade_book[student])

print(f'The average of student totals is {sum_of_grades/4}')
```

Example: Word Counts

```
"""Tokenizing a string and counting unique words."""
text = ('this is sample text with several words '
        'this is more sample text with some different words')
word_counts = {}
# count occurrences of each unique word
for word in text.split():
    if word in word_counts:
        word_counts[word] += 1  # update existing key-value pair
    else:
        word_counts[word] = 1  # insert new key-value pair
print(f'{"WORD":<12}COUNT')
for word, count in sorted(word_counts.items()):
    print(f'{word:<12}{count}')
print('\nNumber of unique words:', len(word_counts))
```

Counter is a customized dictionary

```
from collections import Counter
text = ('this is sample text with several words '
        'this is more sample text with some different words')
counter = Counter(text.split())
for word, count in sorted(counter.items()):
    print(f'{word:<12}{count}')

print('Number of unique keys:', len(counter.keys()))
```

Dictionary Comprehensions

Similar to list comprehensions

for creating a dictionary

Convenient notation for quickly generating dictionaries, often by **mapping** one dictionary to another.

```
myNumbers = {1:1, 2:2, 3:3, 4:4}

mySquares = {key: value**2 for key, value in myNumbers.items()}

print(mySquares)
```

Using zip to create a dictionary

```
names = ['bob', 'ken', 'ron']

grades = [98, 76, 80]

studentGrades = dict(zip(names, grades))

print(studentGrades)
```

Exercise

1. Dictionary

Create a dictionary entry to store customer info: name, Balance and account number. Fill it with a made up customer's info

Sets

Sets

- A set is an unordered collection of **unique values**.
- May contain **only immutable objects**, like strings, ints, floats and tuples that contain only immutable elements.
- Sets do not support indexing and slicing.

```
colors = {'red', 'orange', 'yellow', 'green', 'red', 'blue'}
print(colors)
len(colors)
'red' in colors
for color in colors:
    print(color.upper(), end=' ')
#Creating a Set with the Built-In set Function
numbers = list(range(10)) + list(range(5))
set(numbers)
newSet = set()
```

Create a set from a string

```
x = set('abcd')

print(x)

y = set('This is a test')

print(y)
```

Set operations

Operations in mathematical set theory are supported for the set objects.

The **union** of two sets is a set consisting of all the unique elements from both sets.

```
{1, 3, 5} | {2, 3, 4}
{1, 3, 5}.union([20, 20, 3, 40, 40])
```

The **intersection** of two sets is a set consisting of all the unique elements that the two sets have in common.

```
{1, 3, 5} & {2, 3, 4}
{1, 3, 5}.intersection([1, 2, 2, 3, 3, 4, 4])
```

The **difference** between two sets is a set consisting of the elements in the left operand that are not in the right operand.

```
{1, 3, 5} - {2, 3, 4}
```

The **symmetric difference** between two sets is a set consisting of the elements of both sets that are not in common with one another.

```
{1, 3, 5} ^ {2, 3, 4}
```

Two sets are **disjoint** if they do not have any common elements.

```
{1, 3, 5}.isdisjoint({2, 4, 6})
{1, 3, 5}.isdisjoint({4, 6, 1})
```

Set operations

```
x = set('abcd')
y = set('bdxyz')
x & y
x | y
x-y
set('ab') < x
'e' in x
x.add('r') #for once
x.remove('a')
x.add(1)
x.add((1,2,3))
z = {1,2,3}
type(z)
l1 = [1,2,2,2,3,4]
set(l1)  # will remove duplicates
l1 = list(set(l1)) # will remove duplicates
```