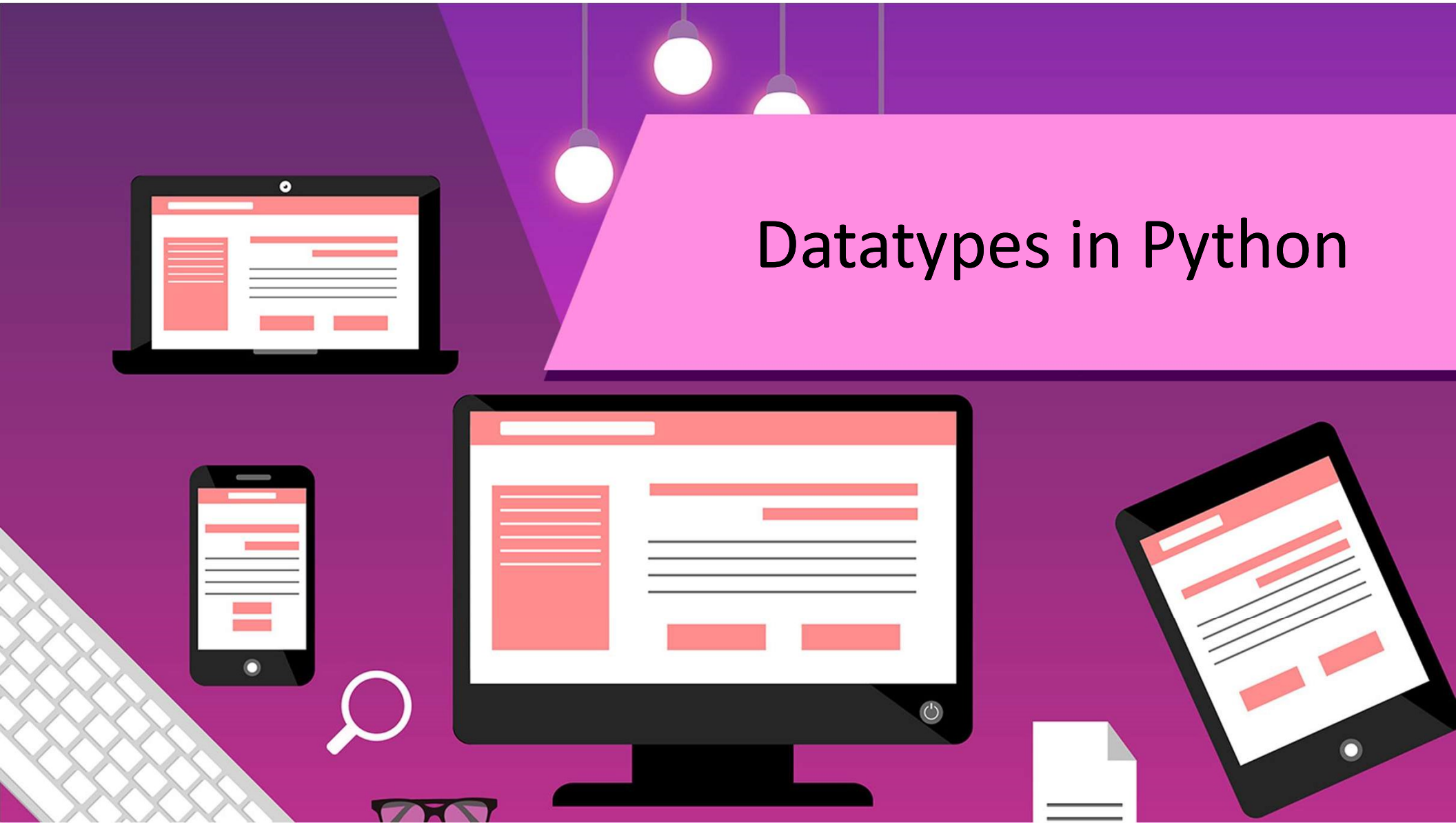
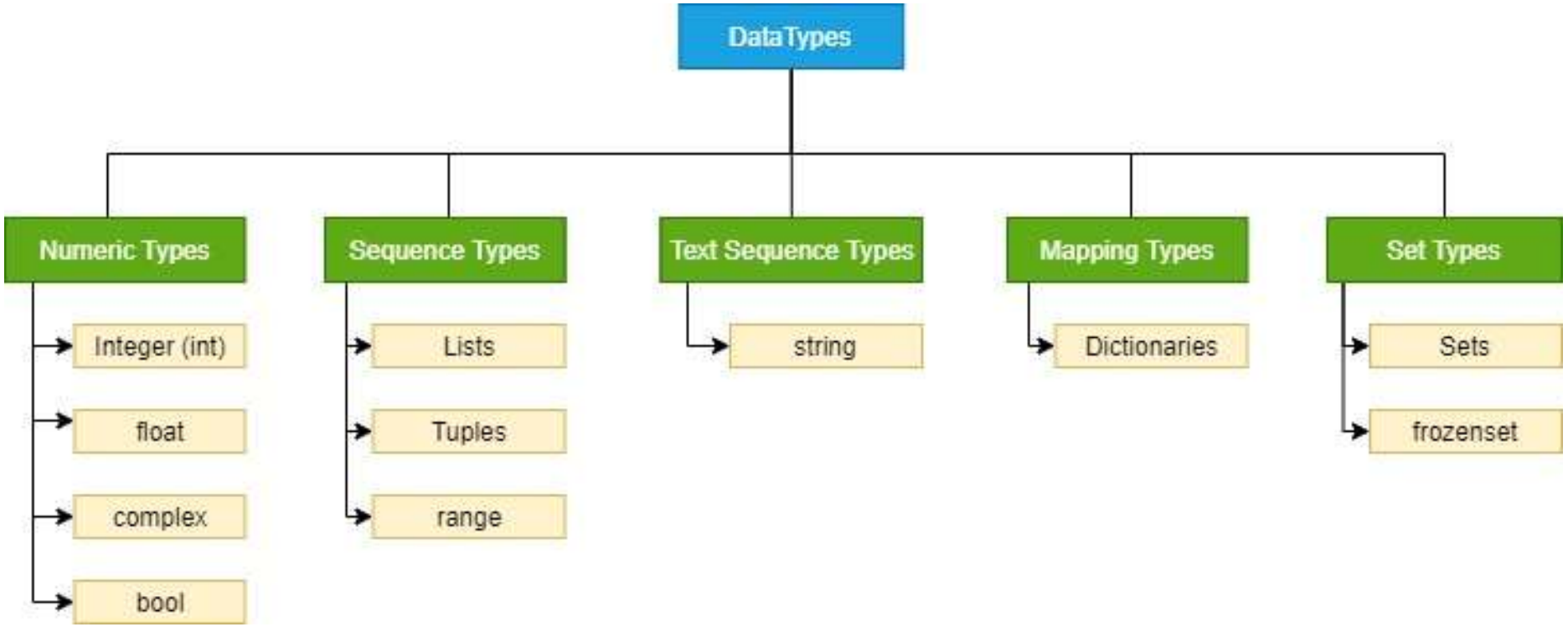

Datatypes in Python

Datatypes in Python



Numeric Types

- **Integer**: int()
  Values without a decimal part
  Ex: x = 3
- **Float**: float()
  Values without a decimal part
  Ex: x = 4.5
- **Complex**: complex(real, imag)
  Values with both a real part and imaginary part
  Ex: x = 2 + 4j
- **Boolean**: bool()
  Can be either True or False only. It is a subtype of integer.
  Ex: x = True, y = False

Text Sequence Types

- **String**:

  A text type value is called a string. It can be a character, word or a sentence. The string value must be enclosed within either ‘’ or “”.

  Ex: name = “Bharat” or name = ‘Bharat’

  Str()

Operators

- There are five types of operators in Python:
  1. Arithmetic Operators
  2. Assignment Operators
  3. Logical/Boolean Operators
  4. Relational/Comparison Operators
  5. Bitwise Operators

Operators

- **Arithmetic Operators**:

  They are addition (**+**), subtraction (**-**), multiplication (*****), division (**/**), exponent (******), floor division (**//**) and modulus (**%**) .

  – Exponent operator raises the power of a number I.e. a**b means a raised to the power of b.

  – Floor division divides a number and the rounds of the result to the lower value.

  – Modulus operator returns the remainder of a division.

  Ex: 5+2, 5*2, 5-2, 5/2, 5**2, 5//2, 5%2

Operators

- **Assignment Operators**:

  Used to assign a value to a variable

  – Basic assignment: =, Ex: x = 56

  – Assignment with an operator: +=, -=, *=, /=, **=, //=, %=

  Ex: a = 5

  a += 8 #Same as a = a + 8

Operators

- **Logical/Boolean Operators**:
  - and, or, not

a	b	a and b
True	True	True
True	False	False
False	True	False
False	False	False

a	b	a or b
True	True	True
True	False	True
False	True	True
False	False	False

a	not a
True	False
False	True

Operators

- **<u>Relational Operators</u>**:
  - Used to compare two values
  - >, <, >=, <=, ==, !=
  - Return Boolean value

  Ex: a<b, a!=b

Operators

- **Bitwise Operators**:
  - Operate bit by bit of the value

Operation	Explanation
x | y	bitwise *or* of *x* and *y*
x ^ y	bitwise *exclusive or* of *x* and *y*
x & y	bitwise *and* of *x* and *y*
x << n	*x* shifted left by *n* bits
x >> n	*x* shifted right by *n* bits
~x	the bits of *x* inverted

A = 5
B = 3
A = 0000 0101 B = 0000 0011
A << 1
A = 0000 1010 = 2 + 8 = 10
n left shift = A* (2**n) n right shift = A/(2**n)
A >> 1
A = 0000 0101 = 5
~A = 1111 1010

Operator Precedence

Symbol	Operator Name
()	Parentheses
**	Exponent
+x, -x, ~x	Unary plus, Unary minus, Bitwise NOT
*, /, //, %	Multiplication, Division, Floor division, Modulus
+, -	Addition, Subtraction
<<, >>	Bitwise shift operators
&	Bitwise AND
^	Bitwise XOR
|	Bitwise OR

Symbol	Operator Name
==, !=, >, >=, <, <=, is, is not, in, not in	Comparisons, Identity, Membership operators
not	Logical NOT
and	Logical AND
or	Logical OR

Text Sequence Types

String Operations:

capitalize()	Converts the first character to upper case
casefold()	Converts string into lower case
center()	Returns a centered string
count()	Returns the number of times a specified value occurs in a string
endswith()	Returns true if the string ends with the specified value
find()	Searches the string for a specified value and returns the position of where it was found
format()	Formats specified values in a string

index()	Searches the string for a specified value and returns the position of where it was found
isalnum()	Returns True if all characters in the string are alphanumeric
isalpha()	Returns True if all characters in the string are in the alphabet
isdecimal()	Returns True if all characters in the string are decimals
isdigit()	Returns True if all characters in the string are digits
isidentifier()	Returns True if the string is an identifier
islower()	Returns True if all characters in the string are lower case
isprintable()	Returns True if all characters in the string are printable
isspace()	Returns True if all characters in the string are whitespaces
istitle()	Returns True if the string follows the rules of a title
isupper()	Returns True if all characters in the string are upper case

Text Sequence Types

join()	Joins the elements of an iterable to the end of the string
ljust()	Returns a left justified version of the string
lower()	Converts a string into lower case
lstrip()	Returns a left trim version of the string
partition()	Returns a tuple where the string is parted into three parts
replace()	Returns a string where a specified value is replaced with a specified value
rfind()	Searches the string for a specified value and returns the last position of where it was found
rindex()	Searches the string for a specified value and returns the last position of where it was found
rjust()	Returns a right justified version of the string
rpartition()	Returns a tuple where the string is parted into three parts

split()	Splits the string at the specified separator, and returns a list
splitlines()	Splits the string at line breaks and returns a list
startswith()	Returns true if the string starts with the specified value
strip()	Returns a trimmed version of the string
swapcase()	Swaps cases, lower case becomes upper case and vice versa
title()	Converts the first character of each word to upper case
upper()	Converts a string into upper case
zfill()	Fills the string with a specified number of 0 values at the beginning

Sequence Types

- Use to store a collection of values of same or different datatypes.
- The different sequence types are:
  - Lists
  - Tuples
  - Range

Lists

- Lists are used to contain a collection of same or different datatypes
- Lists are mutable
- They can be one dimensional or n-dimensional
- List can contain other lists of different datatypes, also called as a “list of lists” or also a “matrix”
- Lists are identified with “[]” brackets

Initialization:

- Lists can be initialized as:
  - Ex: ls = [] or ls = list()

Lists

List Operations:

- append() – add/append a value to the end of a list
- extend()
- insert(i, val)
- index()
- clear()
- pop()
- sort()
- reverse()
- len()
- del
- copy() – Shallow copy
- count()

Tuples

- Tuples are used to contain a collection of same or different datatypes
- Tuples are immutable
- They can be one dimensional or n-dimensional
- Tuples can contain tuples inside them too
- They are identified by a pair of “()” (parenthesis)

Initialization:

- They can be initialized using () or tuple()

  EX: t = (1,2,3), t = tuple((1,2,3))
- Tuple can also be initialized as follows

  Ex: t = 1, 2, 3, ”hello”, (1, 2, 3) (cannot create single element tuple)

Tuples

Tuple Operations:

- index()
- len()
- del
- count()

Unpacking Tuple:

- You can unpack elements in tuple as follows
  - T = (1, 2, 3)
  - a, b, c = T
  - print("{}, {}, {}".format(a, b, c) # Will print 1, 2, 3

Other Tuple operations:

- Allows both positive and negative indexing
- Allows slicing
- Can be concatenated using "+" operator. Ex: (1, 2, 3) + (4, 5, 6) # (1, 2, 3, 4, 5, 6)
- Can use "*" operator for repeated concatenation. Ex: (1, 2, 3) * 2 # (1, 2, 3, 1, 2, 3)

Range()

- Range is an iterator type
- Returns value when accessed(lazy loading)
- The range and steps should always be an integer

Usage:

- X = range(0, 5) # to get numbers from 0 to 4
  - print(X) # outputs range(0, 5)
  - print(X[0]) # outputs 0
- To change step size
  - X = range(0, 5, 2) # to get number from 0 to 4 in steps of two
  - print("{} {}".format(X[0], X[1])) # outputs 0, 2

Mapping Types

- **Dictionaries** contain items as key/value pairs
- Dictionaries allow to specify unique keys to be assigned to values
- Dictionaries are mutable type
- Dictionary is an unordered collection of items
- Dictionary keys can be integers, floating point numbers and strings
- The values can be of any data type

Initialization:

- Can be initialized using either {} or dict()
  - Ex: d = {‘a’:’fdf’, 1:56.89, 3.56:[34, 56]}, d = dict({1:'apple', 2:'ball'}), d = dict([(1, ‘apple’), (2, ‘ball’)])

Dictionaries

Accessing Values:

- d[key] or d.get(key). Ex: d[‘a’], d[1], d.get(‘a)
- d.keys() -> Returns a dict_key list of all the keys
- d.values() -> Returns a dict_values list of all the values
- d.items() -> Returns a list of key-value pair tuples

Updating and Adding values:

- d[key] = value -> Updates the value if the key exists else adds a new key value pair
- d1.update(d2) -> Appends a second dictionary d2 to d1
- {}.fromkeys(list/set/tuple/value, default value) -> Returns a list containing all the specified elements as key initialized with a default value

Dictionaries

Removing items:

- d.pop(key) -> Deletes a specific key-value pair
- d.popitem() -> deletes the last inserted key-value pair in LIFO manner, doesn't accept any arguments
- d.clear() -> Removes all the items from the dictionary
- del d -> Deletes the dictionary

Other operators:

- len(d) -> Returns the number of items present in the dictionary
- sorted(d) -> Returns a sorted **list of keys** of a dictionary

Sets

- Sets contain elements in unordered fashion
- Sets can contain only unique elements
- They are used as a analogy for mathematical sets
- Sets are mutable
- Sets can be used to remove duplicates
- Sets can contain elements of any type

Initialization:

- Sets can be initialized using {} or set() as follows:
  - a = {1, 2, 3, 4} # a will contain only {1, 2, 3, 4}
  - a = {1, 2, 3, 4, 4, 1} # a will contain only {1, 2, 3, 4}
  - A = set([1, 2, 3, 4]) # a will contain only {1, 2, 3, 4}
  - A = {1.0, “hello”, (1, 2, 3)}

Sets

Accessing Values:

- You cannot access individual elements of a set

Adding and Updating Values:

- a.add(element) -> adds the element if it does not existsin set A
- a.update(list/tuple/set) -> adds unique elements of the provided elements to set A
- a.intersection_update(b, c, …..) or a &= b & c …..-> Update the set, keeping only elements found in it and all others
- a.difference_update(b, c…) or a -= b – c … -> Update the set, removing elements found in others
- a.symmetric_difference_update(b, c…) or a ^= b ^ c … -> Update the set, keeping only elements found in either set, but not in both

Removing Elements:

- a.discard(element) -> Removes a elements from the set if it is present
- a.remove(element) -> Removes an element if it exists, otherwise error is thrown
- a.pop() -> Remove the element from the beginning of the list
- a.clear() -> Empty the set
- del a -> delete the set

Sets

Set Operations:

- *a.union(b, c, ..)* or *a | b |* … -> Union of all the sets
- *a.intersection(b, c, …)* or *a & b & c &* ….. -> Intersection of sets
- *a.difference(b, c, ..)* or *a - b - c* -… -> Difference of Sets [Note: a-b != b-a]
- *a.symmetric_difference(b, c, ..)* or *a ^ b ^ c ^*… -> (Union - Intersection) of sets
- *a.isdisjoint(b, c, …)* -> Check if sets are disjoint , returns bool value
- *a.issubset(b, c, …)* or *a < b < c* -> Check if a set is subset of the other
- *a.issuperset(b, c, …)* or *a >= b >= c* -> Check if a set is superset of the other

Others:

- a.copy() -> To create a shallow copy of set a
- i in a - > Check if element i exists in a
- i not in a -> Check if element i exists in a

Frozenset

- Frozensets are similar to sets
- Frozenset is immutable
- All the operations of sets except the update operations can be applied on frozensets

Initialization:

- Frozensets can be initialized using {} or set() as follows:
  - A = frozenset([1, 2, 3, 4]) # a will contain only {1, 2, 3, 4}