Basic Python Notes

Ananth Purohit

Software Engineer

Contents

1 Datatypes

Datatypes are of two types: (i) builtin and (ii) user defined.

User defined datatypes are datatypes that are defined by the user. Builtin datatypes are predefined.

These are the different type of builtin datatypes in Python:

1. Text type: 'str'
2. Numeric types: 'int', 'float', 'complex'
3. Sequence types: 'list', 'tuple', 'range'
4. Mapping type: 'dict'
5. Set type: 'set', 'frozenset'
6. Boolean type: 'bool', 'True'/'False'
7. Binary types: 'bytes', 'bytearray', 'memoryview'
8. None type: 'NoneType' = 'None'

1.1 Python Numbers

Three types: 'int', 'float', and 'complex'.

Random number: Python doesn't have 'random()' to make random numbers, but python has built-in module called 'random' that can be used to make random numbers.

```
1 import random
2 print(random.randrange(0,100))
```

1.2 Python Booleans

There are two main boolean values: 'True' and 'False'. Python has many built-in functions that returns 'bool' value like 'isinstance()' which can determine if an object is of certain datatype or not.

```
1 x = 200
2 print(isinstance(x, int))
```

Output:

```
True
```

2 Python Strings

Surrounded by single or double quote marks: 'hello' = "hello"; both will be considered same by python.

2.1 String operations

2.1.1 upper case

```
a = "Hello World!"
print(a.upper())
```

Output:

```
HELLO WORLD!
```

2.1.2 lower case

```
print(a.lower())
```

Output:

```
hello world!
```

2.1.3 remove white spaces

```
print(a.strip())
```

Output:

```
HelloWorld!
```

2.1.4 replace string

```
print(a.replace(H, L))
```

Output:

```
Lello World!
```

2.1.5 split string

```
print(a.split(" "))
```

Output:

```
["Hello", "World!"]
```

2.1.6 String concatenation

```
a = "Hello"
b = "World"
c = a + b
print(c)
```

Output:

```
HelloWorld
```

2.2 f strings

We can combine strings and numbers by using f-strings.
To specify a string as an f-string, simply put an **f** in front of the string literal, and add curly brackets **{}** as placeholders for variables and other operations.

```
age = 56
txt = "My name is John, I am " + age
print(txt) # This format will throw an error

age = 56
txt = f"My name is John, I am {age}"
print(txt)
```

Output:

```
My name is John, I am 56
```

2.2.1 Placeholders and Modifiers

A placeholder can contain variables, operations, functions, and modifiers to format the value.

```
# Placeholder:
price = 59
txt = f"The price is {price} dollars"
print(txt)

# Modifier:
price = 59
txt = f"The price is {price:.2f} dollars"
print(txt)
```

Output:

```
The price is 59 dollars
The price is 59.00 dollars
```

2.3 Escape characters

To insert characters that are illegal in a string, use an escape character. An escape character is a backslash \ followed by the character you want to insert.

An example of an illegal character is a double quote inside a string that is surrounded by double quotes:

```
txt = "We are the so-called "Vikings" from the north."
```

To fix this problem, use the escape character \":

```
txt = "We are the so-called \"Vikings\" from the north."
```

Code	Result
\'	single quote
\\	backslash
\n	newline
\r	carriage return
\t	tab
\b	back space
\f	form feed
\ooo	octal value
\xhh	hex value

Table 1: Escape Characters

2.4 String methods

Methods	Description
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
encode()	Returns an encoded version of the string
endswith()	Returns true if the string ends with the specified value
expandtabs()	Sets the tab size of the string
find()	Searches the string for a specified value and returns the position of where it was found
format()	Formats specified values in a string
format_map()	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
isascii()	Returns True if all characters in the string are ascii characters
isdecmal()	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
isnumeric()	Returns True if all characters in the string are numeric
isprintable()	Returns True if all characters in the string are printable
ispace()	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
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
maketrance()	Returns a translation table to be used in translations
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
rsplit()	Splits the string at the specified separator, and returns a list
rstrip()	Returns a right trim version of the string
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
translate()	Returns a translated string
upper()	Converts a string into upper case
zfill()	Fills the string with a specified number of 0 values at the beginning

Table 2: String Methods

3 Python Operators

Operators are used to perform operations on variables and values

3.1 Arithmetic operators

Arithmetic operators are used with numeric values to perform common mathematical operations.

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus
**	Exponentiation
//	Floor division

Table 3: Arithmetic operators

3.2 Assignment operators

Assignment operators are used to assign values to variables.

Operator	Example	Same as
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x/= 3	x = x/3
%=	x %= 3	x = x % 3
//=	x //= 3	x = x //3
**=	x **= 3	x = x **3
&=	x &= 3	x = x & 3
| =	x | = 3	x = x | 3
∧=	x ^=3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x >> 3
:=	print(x := 3)	x = 3 print(x)

Table 4: Assignment operators

3.3 Comparison operators

Comparison operators are used to compare two values.

Operator	Name	Example
==	Equal	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Lesser than	x < y
>=	Greater than or equal to	x >= y
<=	Lesser than or equal to	x <= y

Table 5: Comparison operators

3.4 Logical operators

Logical operators are used to combine conditional statements.

Operator	Description	Example
and	Returns True if both the statements are true	x > 5 and x < 10
or	Returns True if one of the statements is true	x < 5 or x < 4
not	Reverse the result, returns False if result is true	not(x > 5 and x < 10)

Table 6: Logical operators

3.5 Identity operators

Identity operators are used to compare the objects, not if they are equal, but if they are actually the same object, with the same memory location.

Operator	Description	Example
is	Returns True if both variables are the same object	x is y
is not	Returns True if both variables are not the same object	x is not y

Table 7: Identity operators

3.6 Membership operators

Membership operators are used to test if a sequence is presented in an object.

Operator	Description	Example
in	Returns True if a sequence with the specified value is present in the object	x in y
not in	Returns True if a sequence with the specified value is not present in the object	x not in y

Table 8: Membership operators

3.7 Bitwise operators

Bitwise operators are used to compare (binary) numbers.

Operator	Name	Description	Example
&	AND	Sets each bit to one if both bits are 1	x & y
|	OR	Set each bit t one if one of two bits is 1	x | y
∧	XOR	Set each bit to one if only one of the two bits is 1	x ∧ y
∼	NOT	Inverts all bits	∼x
<<	Zero fill left shift	Shift left by pushing zeros from the right and let the leftmost bits fall off	x << y
>>	Signed right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off	x >> y

Table 9: Bitwise operators

3.8 Operator precedence

Operator precedence describes the order in which operations are performed.
The precedence order is described in the table below, starting with the highest precedence at the top:

Operator	Description
()	Parentheses
**	Exponentiation
‘+x’ ‘-x’ ‘∼x’	Unary plus, unary minus, and bitwise NOT
‘*’ ‘/’ ‘//’ ‘%’	Multiplication, division, floor division, and modulus
‘+’ ‘-’	Addition and subtraction
‘<<’ ‘>>’	Bitwise left and right shifts
&	Bitwise AND
∧	Bitwise XOR
|	Bitwise OR
‘==’ ‘!=’ ‘>’ ‘>=’ ‘<’ ‘<=’ ‘is’ ‘is not’ ‘in’ ‘not in’	Comparisons, identity, and membership operators
not	Logical NOT
and	AND
or	OR

Table 10: Operator precedence

4 Python List

1. They are used to store multiple items in a single variable.
2. They created by using '[ ]' .
3. Lists are ordered i.e., indexing operations can be performed.
4. They are changeable: we can add, remove items after the lists have been created.
5. Allows duplicates.
6. All data types allowed; that too within the same list.

4.1 Few list operations

4.1.1 Common method of creating a list

```
1 mylist = ["apple", "banana", "cherry"]
```

4.1.2 Alternate method to initiate a list

```
1 a = list(("apple", "banana", "cherry"))
2 print(a)
```

Output:

```
[`apple', `banana', `cherry']
```

4.1.3 Determining list length

we use 'len()'

```
1 mylist = ["apple", "banana", "cherry"]
2 print(len(mylist)) #o/p: 3
```

4.1.4 Changing the elements of a list

method 1:

```
1 mylist = ["apple", "banana", "cherry"]
2 mylist[2] = "coconut"
3 print(mylist)
```

Output:

```
[`apple', `banana', `coconut']
```

method 2:

```
1 mylist.insert(2, "watermelon")
2 print(mylist)
```

Output:

```
[`apple', `banana', `watermelon', `coconut']
```

4.1.5 Adding items at the end of a list

```
mylist.append("durian")
print(mylist)
```

Output:

```
[`apple', `banana', `watermelon', `coconut', `durian']
```

4.1.6 Extending a list

```
mylist = ["apple", "banana", "cherry"]
tropical = ["mango", "pineapple", "papaya"]
mylist.extend(tropical)
print(mylist)
```

Output:

```
[`apple', `banana', `cherry', `mango', `pineapple', `papaya']
```

You can add any iterable objects (like tuple, sets, dictionary) to 'extend()' method

```
mylist = ["apple", "banana", "cherry"]
thistuple = ("kiwi", "orange")
mylist.extend(thistuple)
print(mylist)
```

Output:

```
[`apple', `banana', `cherry', `kiwi', `orange']
```

4.1.7 Remove items from a list

1. remove()

```
mylist = ["apple", "banana", "cherry"]
mylist.remove("cherry")
print(mylist)
```

Output:

```
[`apple', `banana']
```

2. removing at specified index => .pop()

```
mylist = ["apple", "banana", "cherry"]
mylist.pop(1)
print(mylist)

# if index isnt mentioned, it removes last item.
mylist.pop()
print(mylist)
```

Output:

```
[`apple', `cherry']
[`apple']
```

3. del keyword

```
# removing at specific index:
mylist = ["apple", "banana", "cherry"]
del mylist[2]
print(mylist)

# deleting entire list:
del mylist
print(mylist) # o/p: error message as list has been deleted
```

Output:

```
[`apple', `banana']
```

4. clear() method

It empties entire list.

```
mylist = ["apple", "banana", "cherry"]
mylist.clear()
print(mylist) # o/p: []
```

4.1.8 Looping through list

1. for loop:

```
thislist = ["apple", "banana", "cherry"]
for x in thislist:
  print(x)
```

Output:

```
apple
banana
cherry
```

2. Loop through index numbers:

We use the 'range()' and 'len()' functions to create a suitable iterable.

```
thislist = ["apple", "banana", "cherry"]
for i in range(len(thislist)):
  print(thislist[i])
```

Output:

```
apple
banana
cherry
```

3. while loop:

```
thislist = ["apple", "banana", "cherry"]
i = 0
while i < len(thislist):
  print(thislist[i])
  i = i + 1
```

Output:

```
apple
banana
cherry
```

4. Looping using list comprehension:

```
thislist = ["apple", "banana", "cherry"]
[print(x) for x in thislist]
```

Output:

```
apple
banana
cherry
```

4.1.9 List Comprehension

List comprehension offers a shorter syntax when you want to create a new list based on the values of an existing list.

```
# without list comprehension
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
newlist = []

for x in fruits:
  if "a" in x:
    newlist.append(x)

print(newlist)

# with list comprehension
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
newlist = [x for x in fruits if "a" in x]
print(newlist)
```

Output:

```
[`apple', `banana', `mango']
```

4.1.10 Sorting lists

To sort lists, we use 'sort()' method; to obtain the sort in reverse order, we use '.sort(reverse=True)' .

```
thislist = ["orange", "mango", "kiwi", "pineapple", "banana"]
thislist.sort()
print(thislist)

thislist = ["orange", "mango", "kiwi", "pineapple", "banana"]
thislist.sort(reverse = True)
print(thislist)
```

Output:

```
[`banana', `kiwi', `mango', `orange', `pineapple']
[`pineapple', `orange', `mango', `kiwi', `banana']
```

To reverse a list, we use '.reverse()' method.

```
thislist = ["orange", "mango", "kiwi", "pineapple", "banana"]
thislist.reverse()
print(thislist)
```

Output:

```
[`cherry', `Kiwi', `orange', `banana']
```

4.1.11 Copying a list

```
thislist = ["apple", "banana", "cherry"]
mylist = thislist.copy()
print(mylist)

# built-in method:
thislist = ["apple", "banana", "cherry"]
mylist = list(thislist)
print(mylist)
```

Output:

```
[`apple', `banana', `cherry']
[`apple', `banana', `cherry']
```

4.2 List methods

Methods	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 item with the specified value
reverse()	Reverses the order of the list
sort()	Sorts the list

Table 11: List Methods

5 Python Tuple

1. Tuples are used to store multiple items in a single variable.
2. A tuple is a collection which is **ordered** and **unchangeable**.
3. Tuples are written with round brackets '( )'.

5.1 Few tuple operations

Most tuple operations are similar to list operations. Here are few of them:

5.1.1 Creating a tuple

```
1 thistuple = ("apple", "banana", "cherry")
2 print(thistuple)
```

Output:

```
(`apple', `banana', `cherry')
```

5.1.2 tuple length

```
1 thistuple = ("apple", "banana", "cherry")
2 print(len(thistuple)) # o/p: 3
```

5.1.3 Create tuple with one item

```
1 thistuple = ("apple",)
2 print(type(thistuple))
3
4 # NOT a tuple
5 thistuple = ("apple")
6 print(type(thistuple))
```

Output:

```
<class `tuple'>
<class `str'>
```

5.1.4 The tuple() constructor

```
1 thistuple = tuple(("apple", "banana", "cherry"))
2 print(thistuple)
```

Output:

```
(`apple', `banana', `cherry')
```

5.2 tuple methods

Method	Description
count()	Returns the number of times a specified value occurs in a tuple
index()	Searches the tuple for a specified value and returns the position of where it was found

Table 12: tuple methods

6 Python Sets

1. Sets are used to store multiple items in a single variable.
2. A set is a collection which is unordered, and un-indexed.
3. Set items are unchangeable, but you can remove items and add new items.
4. Duplicates not allowed.
5. The values 'True' and '1' are considered the same value in sets, and are treated as duplicates.

6.1 Set operations

6.1.1 Creating a set

```
1 thisset = {"apple", "banana", "cherry"}
2 print(thisset)
```

Output:

```
{`apple', `banana', `cherry'}
```

6.1.2 type()

```
1 myset = {"apple", "banana", "cherry"}
2 print(type(myset))
```

Output:

```
<class `set'>
```

6.1.3 The set() constructor

```
1 thisset = set(("apple", "banana", "cherry"))
2 print(thisset)
```

Output:

```
{`apple', `banana', `cherry'}
```

6.1.4 Adding items

```
1 thisset = {"apple", "banana", "cherry"}
2 thisset.add("orange")
3 print(thisset)
```

Output:

```
{`apple', `banana', `cherry', `orange'}
```

6.1.5 Adding sets

```
1 thisset = {"apple", "banana", "cherry"}
2 tropical = {"pineapple", "mango", "papaya"}
3 thisset.update(tropical)
4 print(thisset)
```

Output:

```
{`apple', `banana', `cherry', `pineapple', `mango', `papaya'}
```

6.1.6 Add any iterable

The object in the 'update()' method does not have to be a set, it can be any iterable object (tuples, lists, dictionaries etc).

```
thisset = {"apple", "banana", "cherry"}
mylist = ["kiwi", "orange"]
thisset.update(mylist)
print(thisset)
```

Output:

```
{`apple', `banana', `cherry', `orange', `kiwi'}
```

6.1.7 Remove item

```
thisset = {"apple", "banana", "cherry"}
thisset.remove("banana") # or thisset.discard("banana")
print(thisset)
```

Output:

```
{`apple', `cherry'}
```

.pop() => Removes a random item

```
thisset = {"apple", "banana", "cherry"}
x = thisset.pop()
print(x)
print(thisset)
```

Output:

```
{`apple', `cherry'}
{`apple', `banana', `cherry'}
```

6.1.8 Join sets

There are several ways to join two or more sets in Python.

1. The 'union()' or '|' and 'update()' methods joins all items from both sets. {can have multiple values}
2. The 'intersection()' or '&' method keeps ONLY the duplicates. {can have multiple values}
3. The 'difference()' or '-' method keeps the items from the first set that are not in the other set(s).
4. The 'symmetric_difference()' method keeps all items EXCEPT the duplicates.

6.2 Set methods

Method	Shortcut	Description
add()		Adds an element to the set
clear()		Removes all the elements from the set
copy()		Returns a copy of the set
difference()	−	Returns a set containing the difference between two or more sets
difference_update()	− =	Removes the items in this set that are also included in another, specified set
discard()		Remove the specified item
intersection()	&	Returns a set, that is the intersection of two other sets
intersection_update()	&=	Removes the items in this set that are not present in other, specified set(s)
isdisjoint()		Returns whether two sets have a intersection or not
issubset()	<=	Returns whether another set contains this set or not
	<	Returns whether all items in this set is present in other, specified set(s)
issuperset()	>=	Returns whether this set contains another set or not
	>	Returns whether all items in other, specified set(s) is present in this set
pop()		Removes an element from the set
remove()		Removes the specified element
symmetric_difference()	∧	Returns a set with the symmetric differences of two sets
symmetric_difference_update()		Inserts the symmetric differences from this set and another
union()	|	Return a set containing the union of sets
update()	| =	Update the set with the union of this set and others

Table 13: Set methods

7 Python Dictionary

1. Dictionaries are used to store data values in key:value pairs.
2. A dictionary is a collection which is ordered*, changeable and do not allow duplicates.
3. Dictionaries are written with curly brackets, and have keys and values.

7.1 Few dictionary operations

7.1.1 Overview

Creating and printing a dictionary

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(thisdict)
```

Output:

```
{`brand': `Ford', `model': `Mustang', `year': 1964}
```

Dictionary items

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(thisdict["brand"]) # o/p: Ford
```

Ordered or Unordered?
As of Python version 3.7, dictionaries are ordered. In Python 3.6 and earlier, dictionaries are unordered.

7.1.2 Accessing items

```
thisdict =  {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}

x = thisdict["model"]
print(x) # o/p: Mustang
```

There is also a method called **get()** that will give you the same result:

```
x = thisdict.get("model")
print(x) # o/p: Mustang
```

The **keys()** method will return a list of all the keys in the dictionary.

```
x = thisdict.keys()
print(x)
```

Output:

```
dict_keys([`brand', `model', `year'])
```

```
car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}

x = car.keys()

# before the change
print(x)

car["color"] = "white"

# after the change
print(x)
```

Output:

```
dict_keys([`brand', `model', `year'])
dict_keys([`brand', `model', `year', `color'])
```

7.1.3 Change items

You can change the value of a specific item by referring to its key name.

```
thisdict =  {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict["year"] = 2018
print(thisdict)
```

Output:

```
{'brand': `Ford', `model': `Mustang', `year': 2018}
```

Update dictionary

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.update({"year": 2020})
print(thisdict)
```

Output:

```
{'brand': `Ford', `model': `Mustang', `year': 2020}
```

7.1.4 Remove items

There are several methods to remove items from a dictionary.

1. **pop()** method removes the item with the specified key name.

```
thisdict =  {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.pop("model")
print(thisdict)
```

Output:

```
{`brand': `Ford', `year': 1964}
```

2. **popitem()** method removes the last inserted item (in versions before 3.7, a random item is removed instead).

```
thisdict =  {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.popitem()
print(thisdict)
```

Output:

```
{`brand': `Ford', `model': `Mustang'}
```

3. **del** keyword removes the item with the specified key name.

```
thisdict =  {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
del thisdict["model"]
print(thisdict)
```

Output:

```
{`brand': `Ford', `model': `Mustang'}
```

The **del** keyword can also delete the dictionary completely.

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
del thisdict
print(thisdict) # this will cause an error because "thisdict" no longer
    exists.
```

4. **clear()** method empties the dictionary.

```
thisdict =  {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.clear()
print(thisdict) # o/p: {}
```

7.1.5 Loop through dictionary

You can use the **values()** method to return values of a dictionary.

```
thisdict =  {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}

for x in thisdict.values():
  print(x)
```

Output:

```
Ford
Mustang
1964
```

You can use the keys() method to return the keys of a dictionary.

```
for x in thisdict.keys():
  print(x)
```

Output:

```
brand
model
year
```

Loop through both keys and values, by using the **items()** method.

```
for x, y in thisdict.items():
  print(x, y)
```

Output:

```
brand Ford
model Mustang
year 1964
```

7.1.6 Nested dictionary

A dictionary can contain dictionaries, this is called nested dictionaries.

```
myfamily = {
  "child1" : {
    "name" : "Emil",
    "year" : 2004
  },
  "child2" : {
    "name" : "Tobias",
    "year" : 2007
  },
  "child3" : {
    "name" : "Linus",
    "year" : 2011
  }
}

print(myfamily)
```

Output:

```
# {`child1': {`name': `Emil', `year': 2004}, `child2': {`name': `Tobias', `year': 2007},
↪  `child3': {`name': `Linus', `year': 2011}}
```

or you can use this method:

```
child1 = {
  "name" : "Emil",
  "year" : 2004
}
child2 = {
  "name" : "Tobias",
  "year" : 2007
}
child3 = {
  "name" : "Linus",
  "year" : 2011
}

myfamily = {
  "child1" : child1,
  "child2" : child2,
  "child3" : child3
}

print(myfamily)
```

Output:

```
{`child1': {`name': `Emil', `year': 2004}, `child2': {`name': `Tobias', `year': 2007},
↪  `child3': {`name': `Linus', `year': 2011}}
```

Accessing items in nested dictionary

```
print(myfamily["child2"]["name"]) # o/p: Tobias
```

Looping through a nested dictionary
You can loop through a dictionary by using the **items()** method

```
for x, obj in myfamily.items():
  print(x)

  for y in obj:
    print(y + `:', obj[y])
```

Output:

```
child1
name: Emil
year: 2004
child2
name: Tobias
year: 2007
child3
name: Linus
year: 2011
```

7.2 Dictionary methods

Modular	Description
clear()	Removes all the elements from the dictionary
copy()	Returns a copy of the dictionary
fromkeys()	Returns a dictionary with the specified keys and value
get()	Returns the value of the specified key
items()	Returns a list containing a tuple for each key value pair
keys()	Returns a list containing the dictionary's keys
pop()	Removes the element with the specified key
popitem()	Removes the last inserted key-value pair
setdefault()	Returns the value of the specified key. If the key does not exist: insert the key, with the specified value
update()	Updates the dictionary with the specified key-value pairs
values	Returns a list of all the values in the dictionary

Table 14: Dictionary Methods

8 Python Conditions (if-elif-else)

Python supports the usual logical conditions from mathematics. These conditions can be used in several ways, most commonly in "if statements" and loops.

8.1 If

An "if statement" is written by using the **if** keyword.

```
if b > a:
  print("b is greater than a")
```

8.2 Elif

The elif keyword is Python's way of saying "if the previous conditions were not true, then try this condition".

```
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
```

8.3 Else

The else keyword catches anything which isn't caught by the preceding conditions.

```
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
else:
  print("a is greater than b")
```

8.4 Shorthand if

If you have only one statement to execute, you can put it on the same line as the if statement.

```
if a > b: print("a is greater than b")
```

8.5 Shorthand if ... else

If you have only one statement to execute, one for if, and one for else, you can put it all on the same line.

```
print("A") if a > b else print("B")
```

8.6 And

The and keyword is a logical operator, and is used to combine conditional statements.

```
if a > b and c > a:
  print("Both conditions are True")
```

8.7 Or

The or keyword is a logical operator, and is used to combine conditional statements.

```
if a > b or a > c:
  print("At least one of the conditions is True")
```

8.8 Not

The not keyword is a logical operator, and is used to reverse the result of the conditional statement.

```
if not a > b:
  print("a is NOT greater than b")
```

8.9 Nested if

You can have 'if' statements inside 'if' statements, this is called nested if statements.

```
if x > 10:
  print("Above ten,")
  if x > 20:
    print("and also above 20!")
  else:
    print("but not above 20.")
```

8.10 pass statement

'if' statements cannot be empty, but if you for some reason have an 'if' statement with no content, put in the 'pass' statement to avoid getting an error.

```
if b > a:
  pass
```

9 Python Loops

Python has two primitive loop commands:

1. while loop
2. for loop

9.1 while loop

With the while loop we can execute a set of statements as long as a condition is true.

```
1 i = 1
2 while i < 6:
3   print(i)
4   i += 1
```

Output:

```
1
2
3
4
5
```

9.1.1 break statement

With the break statement we can stop the loop even if the while condition is true.

```
1 i = 1
2 while i < 6:
3   print(i)
4   if i == 3:
5     break
6   i += 1
```

Output:

```
1
2
3
```

9.1.2 continue statement

With the continue statement we can stop the current iteration, and continue with the next.

```
1 i = 0
2 while i < 6:
3   i += 1
4   if i == 3:
5     continue
6   print(i)
```

Output:

```
1
2
4
5
6
```

9.1.3 else statement

With the else statement we can run a block of code once when the condition no longer is true.

```
i = 1
while i < 6:
  print(i)
  i += 1
else:
  print("i is no longer less than 6")
```

Output:

```
1
2
3
4
5
i is no longer less than 6
```

9.2 for loop

A for loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).

This is less like the for keyword in other programming languages, and works more like an iterator method as found in other object-orientated programming languages.

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  print(x)
```

Output:

```
apple
banana
cherry
```

9.2.1 Looping through a string

Even strings are iterable objects, they contain a sequence of characters.

```
for x in "banana":
  print(x)
```

Output:

```
b
a
n
a
n
a
```

9.2.2 break statement

With the break statement we can stop the loop before it has looped through all the items.

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  print(x)
  if x == "banana":
```

Output:

```
apple
banana
```

9.2.3 continue statement

With the continue statement we can stop the current iteration of the loop, and continue with the next.

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  if x == "banana":
    continue
  print(x)
```

Output:

```
apple
banana
cherry
```

9.2.4 range() function

To loop through a set of code a specified number of times, we can use the range() function.
The range() function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.

```
for x in range(6):
  print(x)
```

Output:

```
0
1
2
3
4
5
```

The range() function defaults to 0 as a starting value, however it is possible to specify the starting value by adding a parameter: range(2, 6), which means values from 2 to 6 (but not including 6).

```
for x in range(2, 6):
  print(x)
```

Output:

```
2
3
4
5
```

The range() function defaults to increment the sequence by 1, however it is possible to specify the increment value by adding a third parameter: range(2, 30, 3)

```
for x in range(2, 30, 3):
  print(x)
```

Output:

```
2
5
8
11
14
17
20
23
26
29
```

9.2.5 else in for loop

The else keyword in a for loop specifies a block of code to be executed when the loop is finished.

```
1 for x in range(6):
2   print(x)
3 else:
4   print("Finally finished!")
```

Output:

```
0
1
2
3
4
5
Finally finished
```

9.2.6 Nested loop

A nested loop is a loop inside a loop. The "inner loop" will be executed one time for each iteration of the "outer loop".

```
1 adj = ["red", "big", "tasty"]
2 fruits = ["apple", "banana", "cherry"]
3
4 for x in adj:
5   for y in fruits:
6     print(x, y)
```

Output:

```
red apple
red banana
red cherry
big apple
big banana
big cherry
tasty apple
tasty banana
tasty cherry
```

9.2.7 pass statement

'for' loops cannot be empty, but if you for some reason have a 'for' loop with no content, put in the pass statement to avoid getting an error.

```
1 for x in [0, 1, 2]:
2   pass
```

10 Python Functions

A function is a block of code which only runs when it is called. You can pass data, known as parameters, into a function. A function can return data as a result.

10.1 Creating a function

In Python a function is defined using the def keyword.

```
1 def my_function():
2   print("Hello from a function")
```

Output:

```
Hello from a function
```

10.2 Calling a function

To call a function, use the function name followed by parenthesis.

```
1 def my_function():
2   print("Hello from a function")
3
4 my_function()
```

Output:

```
Hello from a function
```

10.3 Arguments

Information can be passed into functions as arguments. Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma.

The following example has a function with one argument (fname). When the function is called, we pass along a first name, which is used inside the function to print the full name:

```
1 def my_function(fname):
2   print(fname + " Refsnes")
3
4 my_function("Emil")
5 my_function("Tobias")
6 my_function("Linus")
```

Output:

```
Emil Refsnes
Tobias Refsnes
Linus Refsnes
```

10.3.1 Parameters or Arguments?

The terms parameter and argument can be used for the same thing: information that are passed into a function.

From a function's perspective:

1. A parameter is the variable listed inside the parentheses in the function definition.
2. An argument is the value that is sent to the function when it is called.

10.3.2 Number of arguments

By default, a function must be called with the correct number of arguments. Meaning that if your function expects 2 arguments, you have to call the function with 2 arguments, not more, and not less.

10.3.3 Arbitrary arguments (*args)

If you do not know how many arguments that will be passed into your function, add a * before the parameter name in the function definition

```
def my_function(*kids):
  print("The youngest child is " + kids[2])

my_function("Emil", "Tobias", "Linus")
```

Output:

```
The youngest child is Linus
```

10.3.4 Arbitrary keyword arguments (**kwargs)

If you do not know how many keyword arguments that will be passed into your function, add two asterisk: ** before the parameter name in the function definition. This way the function will receive a dictionary of arguments, and can access the items accordingly.

```
def my_function(**kid):
  print("His last name is " + kid["lname"])

my_function(fname = "Tobias", lname = "Refsnes")
```

Output:

```
His last name is Refsnes
```

10.3.5 Default parameter value

If we call the function without argument, it uses the default value

```
def my_function(country = "Norway"):
  print("I am from " + country)

my_function("Sweden")
my_function("India")
my_function()
my_function("Brazil")
```

Output:

```
I am from Sweden
I am from India
I am from Norway
I am from Brazil
```

10.3.6 Passing a List as an Argument

You can send any data types of argument to a function (string, number, list, dictionary etc.), and it will be treated as the same data type inside the function.

```
def my_function(food):
  for x in food:
    print(x)

```

```
fruits = ["apple", "banana", "cherry"]

my_function(fruits)
```

Output:

```
apple
banana
cherry
```

10.3.7 Return values

To let a function return a value, use the return statement.

```
def my_function(x):
  return 5 * x

print(my_function(3))
print(my_function(5))
print(my_function(9))
```

Output:

```
15
25
45
```

10.3.8 pass statement

function definitions cannot be empty, but if you for some reason have a function definition with no content, put in the pass statement to avoid getting an error.

```
def myfunction():
  pass

# having an empty function definition like this, would raise an error
    without the pass statement
```

10.3.9 Positional arguments

Positional arguments are matched to function parameters based on their position (order) in the function call. When you call a function, the first argument you pass is assigned to the first parameter, the second argument to the second parameter, and so on.

```
def greet(first_name, last_name):
    print(f"Hello, {first_name} {last_name}!")

greet("John", "Doe")  # Positional arguments
```

1. 'first_name' is assigned the value "John".
2. 'last_name' is assigned the value "Doe".

In this case, the function call greet("John", "Doe") uses positional arguments because the values "John" and "Doe" are passed in order to the parameters first_name and last_name, respectively.

To specify that a function can have only positional arguments, you can add '/' after the arguments:

```
def my_function(x, /):
  print(x)

my_function(3) # o/p: 3
```

Without the , / you are actually allowed to use keyword arguments even if the function expects positional arguments:

```
def my_function(x):
  print(x)

my_function(x = 3)  # o/p: 3
```

But when adding the , / you will get an error if you try to send a keyword argument:

```
def my_function(*, x):
  print(x)

my_function(x = 3) # o/p: 3
```

10.3.10 Keyword arguments

Keyword arguments are matched to function parameters by their names. When calling a function, you can explicitly specify which value goes to which parameter by using the parameter names.

```
def greet(first_name, last_name):
    print(f"Hello, {first_name} {last_name}!")

greet(first_name="John", last_name="Doe")  # Keyword arguments
```

To specify that a function can have only keyword arguments, you can add *, before the arguments:

```
def my_function(*, x):
  print(x)

my_function(x = 3)
```

Without the *, you are allowed to use positional arguments even if the function expects keyword arguments:

```
def my_function(x):
  print(x)

my_function(x = 3)
```

But when adding the *, / you will get an error if you try to send a positional argument:

```
def my_function(*, x):
  print(x)

my_function(3)
```

10.3.11 Combining positional and keyword arguments

You can combine the two argument types in the same function. Any argument before the / , are positional-only, and any argument after the *, are keyword-only.

```
def my_function(a, b, /, *, c, d):
  print(a + b + c + d)

my_function(5, 6, c = 7, d = 8) # o/p: 26
```

Another example:

```
def greet(first_name, last_name):
    print(f"Hello, {first_name} {last_name}!")

greet("John", last_name="Doe")  # Mixing positional and keyword arguments
```

10.3.12 Difference between positional and keyword arguments

1. Order:
   (a) Positional Arguments: The order matters. The first argument matches the first parameter, the second argument matches the second parameter, and so on.
   (b) Keyword Arguments: The order does not matter. Each argument is matched to the parameter with the corresponding name.
2. Readability:
   (a) Positional Arguments: Can be less readable, especially if there are many arguments or the function parameters are not self-explanatory.
   (b) Keyword Arguments: Can improve readability by explicitly stating which value is assigned to which parameter.
3. Default Values:
   (a) Positional Arguments: You must provide all preceding arguments if you want to skip to a later one.
   (b) Keyword Arguments: You can skip arguments with default values by specifying only the ones you need.

Conclusion:

1. Positional Arguments: Matched by position; order matters.
2. Keyword Arguments: Matched by name; order does not matter; can enhance readability and flexibility.

10.4 Recursive functions a.k.a Recursion

A function which calls itself is known as recursive function. Let us understand this with the help of factorial example.

Say suppose you want to find the factorial of number 5. The way you go about is: $5! = 5 * 4 * 3 * 2 * 1$. To implement this in programming, you can either use a for loop or use a recursive function. The resursive function code would look something like this:

```
1  def factorial(n):
2      if n == 1:
3          return result  = 1
4      else:
5          result = n * factorial(n - 1)
6      return result
7
8  n = int(input("Enter the number you want to find the factorial of: "))
9  factorial(n)
10 print("The value of {}! is {}.".format(n, result))
```

Part 1 of the output would be:

```
Enter the number you want to find the factorial of: 5
```

The flow of the program would be:

1. The program prompts you to enter the number you want to want the factorial of.
2. This triggers the factorial function.
3. Next it checks for the condition if the value of n is 1. If the condition is true, it returns the value of result to be one.
   This is important as this condition acts as an exit case for recursion; or else the machine would hang as it would enter an infinite loop.
4. The flow of recursion would be:

i. The program would try to find the result, which is 5!

```
result = 5 * factorial(4)
```

ii. Since it doesn't know the value of value of factorial(4), the factorial function is called again and will proceed to find the value of factorial(4).

```
factorial(4) = 4 * factorial(3)
```

iii. This function calling - case checking process continues until we have reached n == 0.

```
factorial(3) = 3 * factorial(2)
factorial(2) = 2 * factorial(1)
factorial(1) = 1 * factorial(0)
```

iv. When the factorial function is called again to find the value of factorial(0), the if case becomes true and will return the factorial value to be 1.
Using this value, the function backtracks to find the value of the original result, 5!.

```
result = 5 * 4 * 3 * 2 * 1 * 1
factorial(4) = 4 * 3 * 2 * 1 * 1
factorial(3) = 3 * 2 * 1 * 1
factorial(2) = 2 * 1 * 1
factorial(1) = 1 * 1
```

5. So, the factorial() function would return the value of result as 120.

The final output would look like:

```
Enter the number you want to find the factorial of: 5
The value of 5! is 120.
```

10.5 Anonymous functions or Lambdas

A lambda function is a small anonymous function. A lambda function can take any number of arguments, but can only have one expression.

```
x = lambda a: a + 10
print(x(5)) # o/p: 15
```

Lambda functions can take any number of arguments.

```
x = lambda a, b : a * b
print(x(5, 6)) # o/p: 30
```

10.5.1 Why use lambda functions?

The power of lambda is better shown when you use them as an anonymous function inside another function. Say you have a function definition that takes one argument, and that argument will be multiplied with an unknown number:

```
def myfunc(n):
  return lambda a : a * n
```

Example: Use that function definition to make a function that always doubles or triples the number you send in

```
def myfunc(n):
  return lambda a : a * n

mydoubler = myfunc(2)
mytripler = myfunc(3)

print(mydoubler(11) # o/p: 22
print(mydoubler(11) # o/p: 33
```

Another example to find the bigger number of the two:

```
max = lambda x, y : x if x>y else y
a, b = [int(n) for n in input("Enter two numbers: "),split(',')]
print("Bigger number: ",max(a, b)) # calling lambda function
```

Output 1:

```
Enter two numbers: 10,20
Bigger number: 20
```

Output 2:

```
Enter two numbers: 20,10
Bigger number: 20
```

10.5.2 Using lambdas with filter() function

The filter() function is useful to filter out the elements of a sequence depending on the result of a function.

Format of filter() function:

```
filter(function, sequence)
```

Here,

1. The 'function' represents a function name that may return either 'True' or 'False' value.
2. 'sequence' represents a list, string or tuple.

Example (without lambda):

```
def is_even(x):
    if x%2==0:
        return True
    else:
    return False
lst = [10, 23, 45, 46, 70, 99]
lst1 = list(filter(is_even, lst))
print(lst1)
```

Output:

```
[10, 46, 70]
```

Example (with lambda):

```
lst = [10, 23, 45, 46, 70, 99]
lst1 = list(filter(lambda x : (x%2==0), lst))
print(lst1)
```

Output:

```
[10, 46, 70]
```

10.5.3 Using lambdas with map() function

The map() is similar to filter() function, but acts on each element of the sequence and perhaps changes the elements

Format of map() function:

```
map(function, sequence)
```

A lambda that returns squares of elements in a list:

```
1 lst = [1, 2, 3, 4, 5]
2 lst1 = list(map(lambda x : x * x, lst))
3 print(lst1)
```

Output:

```
[1, 4, 9, 16, 25]
```

It is possible to use map() for more than one list, if the lists are of similar length.

```
1 lst1 = [1, 2, 3, 4, 5]
2 lst2 = [10, 20, 30, 40, 50]
3 lst3 = list(map(lambda x, y : x * y, lst1, lst2))
4 print(lst3)
```

Output:

```
[10, 40, 90, 160, 250]
```

10.5.4 Using lambdas with reduce() function

The reduce() function reduces a sequence of elements to a single value by processing the elements according to a function supplied.

Format of reduce() function:

```
reduce(function, sequence)
```

Example:

```
1 from functools import *
2 lst = [1, 2, 3, 4, 5]
3 result = reduce(lambda x,y : x * y, lst)
4 print(result)
```

Output:

```
120
```

Another example:

```
1 sum = reduce(lambda a, b : a + b, range(1, 51))
2 print(sum)
```

Output:

```
1245
```

10.6 Function decorators

Decorator is a function that accepts a function as parameter and returns a function. A decorator takes the result of a function, modifies the result and returns it. Thus, decorators are useful to perform some additional processing required by a function.

The following steps are genera;;y involved in the creation of decorators:

1. We should define a function with another function name as parameter.
   As an example, let's define a decorator function decor with fun as parameter.

```
def decor(fun):
```

2. We should define a function inside the decorator function, This function actually modifies or decorates the value of the function passed to the decorator function.
   As a n example, let's write inner() function in the decor() function. Our assumption is that this function increases the value returned by the function by 2.

```
def decor(fun):
    def inner():
        value = fun() # access value returned by fun()
        return value + 2 # increase the value by 2
    return inner # return the inner function
```

3. Return the inner function that has been processed or decorated the value. (this has already been done in the last step of the programming example)

4. Once the decorator is created, it can be used for any function to decorate or process it's result. For example:

```
def num():
    return 10
```

5. Now, we call the decor() function by passing num() function name as:

```
result_fun  = decor(num)
```

   The name 'num' is copied into the parameter of the decor() function. Thus, 'fun' refers to 'num'.

6. In the inner(), value represents 10 and is incremented by 2. The returned function 'inner' is referenced by 'result_fun'. So, calling this function and print the result as:

```
print(result_fun())
```

7. This will display 12.

The complete program will look like this:

```
def decor(fun): # this is a decorator function
    def inner(): # this is the inner function that modifies
        value = fun()
        return value + 2
    return inner # return the inner function

# take a function to whch decorator has to be applied to:
def num():
    return 10

# call the deocrator function and pass num
result_fun = decor(num) # result_fun represents the 'inner' function
print(result_fun()) # call result_fun and display the result
```

Output:

```
12
```

10.7 Generators

Generators are functions that return a sequence of values. A generator function is written as an arbitrary function, but it uses 'yield' statement. This statement is used to return the value.

Example: A generator that returns sequence of numbers from x to y

```
def mygen(x, y): # our generator name is 'mygen'
    while x<= y: # this is loop that repeats from x to y
        yield x # return the x value
        x+=1 # increment x value by 1

# fill generator objects with 5 and 10
g = mygen(5, 10)

# display all the numbers in the generator
for i in g:
    print(i, end=' ')
```

Output:

```
5 6 7 8 9 10
```

10.8 Creating own modules in python

A module represents a group of classes, methods, functions, and variables. While we are developing software, there may be several classes, methods, and functions. We should first group them depending on their depending on their relationship with into various modules and later use these modules in other programs.

Creating our very own module name employee:

```
# calculate dearness allowance:
def da(basic):
    ''' DA is 80% of basic salary. '''

    da = basic * 80/100
    return da

# calculate house rent allowance:
def hra(basic):
    ''' HRA is 15% of basic salary. '''

    hra = basic * 15/100
    rerurn hra

# calculate provident fund amount:
def pf(basic):
    ''' PF is 12% of basic salary. '''

    pf = basic * 12/100
    return pf

# calculate income tax paybale by the employee:
def itax(basic):
    ''' tax is calculated at 10% on gross '''

    tax = gross * 0.1
    return itax
```

Save this file as 'employee.py'.

Using this module in a program:

```
from employee import *

basic = float(input('Enter your basic salary: '))

# calculate gross salary:
gross = basic + da(basic) + hra(basic)
print('Your gross salary: {:10.2f}'.format(gross))

# calculate net salary:
net = gross - pf(basic) - itax(basic)
print('Your gross net: {:10.2f}'.format(net))
```

Output:

```
Enter your basic salary: 15000
Your gross salary: 29250.00
Your net salary: 24525.00
```