Unit_1

August 22, 2023

1 Variable in Python

[2]:
```
#creating variable
x = 5
y = "Aquib"

print(x)
print(y)
```

```
5
Aquib
```

[ ]:
```
Variables do not need to be declared with any particular type, and can even␣
 ↪change type after they have been set.
```

[4]:
```
x = 4
x = "Aquib"
x = 7.5
print(x)
```

```
7.5
```

[ ]:
```
Casting
If you want to specify the data type of a variable, this can be done with␣
 ↪casting.
```

[5]:
```
x = str(3)
y = int(3)
z = float(5)

print(x)
print(y)
print(z)
```

```
3
3
5.0
```

[ ]:
```
Get the Type
You can get the data type of a variable with the type() function.
```

[6]:
```
x = 5
y = "Aquib"

print(type(x))
print(type(y))
```

```
<class 'int'>
<class 'str'>
```

[ ]:
```
Single or Double Quotes?
String variables can be declared either by using single or double quotes:
```

[7]:
```
x = 'Aquib'    # it is same as
y = "Aquib"

print(x)
print(y)
```

```
Aquib
Aquib
```

[ ]:
```
Case-Sensitive
Variable names are case-sensitive.
```

[8]:
```
a = 7
A = "Aquib"
print(a)
print(A)
```

```
7
Aquib
```

[ ]:
```
Variable Names
A variable name must start with a letter or the underscore character
A variable name cannot start with a number
A variable name can only contain alpha-numeric characters and underscores (A-z,␣
 ↪0-9, and _ )
Variable names are case-sensitive (age, Age and AGE are three different␣
 ↪variables)
A variable name cannot be any of the Python keywords.
```

[10]:
```
# Legal variable names:

myvar = "Aquib"
my_var = "Aquib"
_my_var = "Aquib"
```

```
myVar = "Aquib"
MYVAR = "Aquib"
myvar2 = "Aquib"


print(myvar)
print(my_var)
print(_my_var)
print(myVar)
print(MYVAR)
print(myvar2)
```

```
Aquib
Aquib
Aquib
Aquib
Aquib
Aquib
```

[11]:
```
#Illegal variable names:

2myvar = "Aquib"
my-var = "Aquib"
my var = "Aquib"
print(2myvar)
print(my-var)
print(my var)
```

```
  Cell In[11], line 3
    2myvar = "Aquib"
    ^
SyntaxError: invalid decimal literal
```

[ ]:
```
Remember that variable names are case-sensitive
```

[ ]:
```
Multi Words Variable Names
Variable names with more than one word can be difficult to read.

There are several techniques you can use to make them more readable:

Camel Case
Each word, except the first, starts with a capital letter:
```

[12]:
```
myVariableName = "Aquib"
```

[ ]:
```
Pascal Case
Each word starts with a capital letter:
```

[13]:
```
MyVarriableName = "Aquib"
```

[ ]:
```
Snake Case
Each word is separated by an underscore character:
```

[ ]:
```
my_variable_name = "Aquib"
```

[ ]:
```
Many Values to Multiple Variables
Python allows you to assign values to multiple variables in one line:
```

[14]:
```
x,y,z = "Aquib", "Aiyaz", "Aarohi"
print(x)
print(y)
print(z)
```

```
Aquib
Aiyaz
Aarohi
```

[ ]:
```
Note: Make sure the number of variables matches the number of values, or else␣
↪you will get an error.
```

[ ]:
```
One Value to Multiple Variables
And you can assign the same value to multiple variables in one line:
```

[15]:
```
x = y = z = "Aquib Sheikh"
print(x)
print(y)
print(z)
```

```
Aquib Sheikh
Aquib Sheikh
Aquib Sheikh
```

[ ]:
```
Unpack a Collection
If you have a collection of values in a list, tuple etc. Python allows you to␣
↪extract the values into variables.
This is called unpacking.
```

[16]:
```
friendship = ["Aquib", "Aiyaz","Aarohi"]
x, y, z = friendship
print(x)
print(y)
print(z)
```

```
Aquib
Aiyaz
Aarohi
```

[ ]:
```
Output Variables
The Python print() function is often used to output variables.
```

[17]:
```
x = "Aquib is the best"
print(x)
```

```
Aquib is the best
```

[ ]:
```
In the print() function, you output multiple variables, separated by a comma:
```

[19]:
```
x = "Aquib"
y = "is the"
z = "best"

print(x,y,z)
```

```
Aquib is the best
```

[ ]:
```
You can also use the + operator to output multiple variables:
```

[21]:
```
x = "Aquib"
y = " is"
z = " innocent"
print(x+y+z)
```

```
Aquib is innocent
```

[ ]:
```
For numbers, the + character works as a mathematical operator:
```

[22]:
```
x = 10
y = 40
print(x+y)
```

```
50
```

[ ]:
```
In the print() function, when you try to combine a string and a number with the␣
↪+ operator, Python will give you an error:
```

[23]:
```
x = 5
y = "Aquib"
print(x+y)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[23], line 3
      1 x = 5
```

```
      2 y = "Aquib"
----> 3 print(x+y)

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

[ ]:
```
The best way to output multiple variables in the print() function is to␣
 ↪separate them with commas
which even support different data types:
```

[25]:
```
x = 1
y = "Aquib"
print(x,y)
```

```
1 Aquib
```

[ ]:
```
Global Variables
Variables that are created outside of a function (as in all of the examples␣
 ↪above) are known as global variables.

Global variables can be used by everyone, both inside of functions and outside.
```

[42]:
```
x = "awesome"
def myfunc():
     print("Aquib is " + x)
myfunc()
```

```
Aquib is awesome
```

[ ]:
```
If you create a variable with the same name inside a function, this variable␣
 ↪will be local,
and can only be used inside the function. The global variable with the same␣
 ↪name will remain as it was, global
and with the original value.
```

[43]:
```
x = "awesome"
def myfunc():
    x = "fantastic"
    print("Aquib is " + x)
myfunc()
print("Aquib is " + x)
```

```
Aquib is fantastic
Aquib is awesome
```

[ ]:
```
The global Keyword
```

```
Normally, when you create a variable inside a function, that variable is local,
 ↪and can only be used inside that function.

To create a global variable inside a function, you can use the global keyword.
```

[44]:
```
def myfunc():
    global x
    x = "awesome"
    print("Aquib is " + x)
myfunc()
print("Aiyaz is " + x)
```

```
Aquib is awesome
Aiyaz is awesome
```

[ ]:
```
Also, use the global keyword if you want to change a global variable inside a
 ↪function.
```

[45]:
```
x = "awesome"

def myfunc():
    global x
    x = "fantastic"
myfunc()
print("Aquib is "+ x)
```

```
Aquib is fantastic
```

2 Python Data Types

[ ]:
```
Built-in Data Types
In programming, data type is an important concept.

Variables can store data of different types, and different types can do
 ↪different things.

Python has the following data types built-in by default, in these categories:

Text Type:        str
Numeric Types:        int, float, complex
Sequence Types:        list, tuple, range
Mapping Type:        dict
Set Types:        set, frozenset
Boolean Type:        bool
Binary Types:        bytes, bytearray, memoryview
None Type:        NoneType
```

Getting the Data Type You can get the data type of any object by using the type() function:

[56]:
```
x1 = 5
x2 = "Aquib"
x3 = 5.5
x4 = 2j
x5 = ["Aquib","Aiyaz","Anonimous"]
x6 = ("Aquib","Aiyaz","Unknown")
x7 = range(9)
x8 = {"name" : "Aquib", "Age" : 23}
x9 = {"Aquib","Aiyaz","unknown"}
x10 = frozenset({"Aquib","Aiyaz","Unknown"})
x11 = b"Aquib"
x12 = bytearray(5)
x13 = memoryview(bytes(5))
x14 = None
x15 = True
print(type(x1))
print(type(x2))
print(type(x3))
print(type(x4))
print(type(x5))
print(type(x6))
print(type(x7))
print(type(x8))
print(type(x9))
print(type(x10))
print(type(x11))
print(type(x12))
print(type(x13))
print(type(x14))
print(type(x15))
```

```
<class 'int'>
<class 'str'>
<class 'float'>
<class 'complex'>
<class 'list'>
<class 'tuple'>
<class 'range'>
<class 'dict'>
<class 'set'>
<class 'frozenset'>
<class 'bytes'>
<class 'bytearray'>
<class 'memoryview'>
<class 'NoneType'>
<class 'bool'>
```

3 Python Numbers

[ ]:
```
Python Numbers
There are three numeric types in Python:

int
float
complex
Variables of numeric types are created when you assign a value to them:
```

[57]:
```
x = 2
y = 2.2
z = 2j
print(type(x))
print(type(y))
print(type(z))
```

```
<class 'int'>
<class 'float'>
<class 'complex'>
```

[ ]:
```
Int
Int, or integer, is a whole number, positive or negative, without decimals, of␣
 ↪unlimited length
```

[58]:
```
x = 1
y = 35656222554887711
z = -3255522

print(type(x))
print(type(y))
print(type(z))
```

```
<class 'int'>
<class 'int'>
<class 'int'>
```

[ ]:
```
Float
Float, or "floating point number" is a number, positive or negative, containing␣
 ↪one or more decimals.
```

[59]:
```
x = 1.10
y = 1.0
z = -35.59

print(type(x))
print(type(y))
print(type(z))
```

```
<class 'float'>
<class 'float'>
<class 'float'>
```

[ ]:
```
Float can also be scientific numbers with an "e" to indicate the power of 10.
```

[60]:
```
x = 35e3
y = 12E4
z = -87.7e100

print(type(x))
print(type(y))
print(type(z))
```

```
<class 'float'>
<class 'float'>
<class 'float'>
```

[ ]:
```
Complex
Complex numbers are written with a "j" as the imaginary part:
```

[61]:
```
x = 3+5j
y = 5j
z = -5j

print(type(x))
print(type(y))
print(type(z))
```

```
<class 'complex'>
<class 'complex'>
<class 'complex'>
```

[ ]:
```
Type Conversion
You can convert from one type to another with the int(), float(), and complex()␣
 ↪methods:
```

[62]:
```
x = 1    # int
y = 2.8  # float
z = 1j   # complex

#convert from int to float:
a = float(x)

#convert from float to int:
b = int(y)

#convert from int to complex:
c = complex(x)
```

```
print(a)
print(b)
print(c)

print(type(a))
print(type(b))
print(type(c))
```

```
1.0
2
(1+0j)
<class 'float'>
<class 'int'>
<class 'complex'>
```

[ ]:
```
Note: You cannot convert complex numbers into another number type.
```

[ ]:
```
Random Number
Python does not have a random() function to make a random number,
but Python has a built-in module called random that can be used to make random␣
 ↪numbers:
```

[63]:
```
import random

print(random.randrange(1, 10))
```

```
9
```

4 Python Casting

[ ]:
```
Specify a Variable Type
There may be times when you want to specify a type on to a variable. This can␣
 ↪be done with casting.
Python is an object-orientated language,
and as such it uses classes to define data types, including its primitive types.

Casting in python is therefore done using constructor functions:

int() - constructs an integer number from an integer literal, a float literal␣
 ↪(by removing all decimals), or a string literal
(providing the string represents a whole number)
float() - constructs a float number from an integer literal, a float literal or␣
 ↪a string literal
(providing the string represents a float or an integer)
str() - constructs a string from a wide variety of data types, including␣
 ↪strings, integer literals and float literals
```

[ ]:
```
x = int(1)   # x will be 1
y = int(2.8) # y will be 2
z = int("3") # z will be 3

x = float(1)     # x will be 1.0
y = float(2.8)   # y will be 2.8
z = float("3")   # z will be 3.0
w = float("4.2") # w will be 4.2

x = str("s1") # x will be 's1'
y = str(2)    # y will be '2'
z = str(3.0)  # z will be '3.0'
```

5 Python String

[ ]:
```
Strings
Strings in python are surrounded by either single quotation marks, or double␣
 ↪quotation marks.

'hello' is the same as "hello".

You can display a string literal with the print() function:
```

[64]:
```
print("Hello")
print('Hello')
```

```
Hello
Hello
```

[ ]:
```
Assign String to a Variable
Assigning a string to a variable is done with the variable name followed by an␣
 ↪equal sign and the string:
```

[65]:
```
a = "Hello"
print(a)
```

```
Hello
```

[ ]:
```
Multiline Strings
You can assign a multiline string to a variable by using three quotes:
```

[66]:
```
a = """Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua."""
print(a)
```

```
Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua.
```

[ ]:
```
Or three single quotes:
```

[67]:
```
a = '''Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua.'''
print(a)
```

```
Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua.
```

[ ]:
```
Note: in the result, the line breaks are inserted at the same position as in␣
 ↪the code.
```

[ ]:
```
Strings are Arrays
Like many other popular programming languages, strings in Python are arrays of␣
 ↪bytes representing unicode characters.

However, Python does not have a character data type, a single character is␣
 ↪simply a string with a length of 1.

Square brackets can be used to access elements of the string.
```

[68]:
```
a = "Hello, World!"
print(a[1])
```

```
e
```

[ ]:
```
Looping Through a String
Since strings are arrays, we can loop through the characters in a string, with␣
 ↪a for loop.
```

[69]:
```
for x in "banana":
  print(x)
```

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

[ ]:
```
String Length
To get the length of a string, use the len() function.
```

[70]:
```
a = "Hello, World!"
print(len(a))
```

```
13
```

[ ]:
```
Check String
To check if a certain phrase or character is present in a string, we can use␣
 ↪the keyword in.
```

[71]:
```
txt = "The best things in life are free!"
print("free" in txt)
```

```
True
```

[ ]:
```
Use it in an if statement:
```

[72]:
```
txt = "The best things in life are free!"
if "free" in txt:
  print("Yes, 'free' is present.")
```

```
Yes, 'free' is present.
```

[ ]:
```
Check if NOT
To check if a certain phrase or character is NOT present in a string, we can␣
 ↪use the keyword not in.
```

[73]:
```
txt = "The best things in life are free!"
print("expensive" not in txt)
```

```
True
```

[ ]:
```
Use it in an if statement:
```

[74]:
```
txt = "The best things in life are free!"
if "expensive" not in txt:
  print("No, 'expensive' is NOT present.")
```

```
No, 'expensive' is NOT present.
```

[ ]:
```
Slicing
You can return a range of characters by using the slice syntax.

Specify the start index and the end index, separated by a colon, to return a␣
 ↪part of the string.
```

[75]:
```
#Get the characters from position 2 to position 5 (not included):

```

```
b = "Hello, World!"
print(b[2:5])
```

```
llo
```

[ ]:
```
Slice From the Start
By leaving out the start index, the range will start at the first character:
```

[76]:
```
#Get the characters from the start to position 5 (not included):

b = "Hello, World!"
print(b[:5])
```

```
Hello
```

[ ]:
```
Slice To the End
By leaving out the end index, the range will go to the end:
```

[77]:
```
#Get the characters from position 2, and all the way to the end:

b = "Hello, World!"
print(b[2:])
```

```
llo, World!
```

[ ]:
```
Negative Indexing
Use negative indexes to start the slice from the end of the string:
```

[78]:
```
b = "Hello, World!"
print(b[-5:-2])
```

```
orl
```

[ ]:
```
Python has a set of built-in methods that you can use on strings.

Upper Case
```

[79]:
```
a = "Hello, World!"
print(a.upper())
```

```
HELLO, WORLD!
```

[ ]:
```
Lower Case
```

[80]:
```
a = "Hello, World!"
print(a.lower())
```

```
hello, world!
```

[ ]:
```
Remove Whitespace
Whitespace is the space before and/or after the actual text, and very often you␣
 ↪want to remove this space.
```

[81]:
```
a = " Hello, World! "
print(a.strip()) # returns "Hello, World!"
```

```
Hello, World!
```

[ ]:
```
Replace String
```

[82]:
```
a = "Hello, World!"
print(a.replace("H", "J"))
```

```
Jello, World!
```

[ ]:
```
Split String
The split() method returns a list where the text between the specified␣
 ↪separator becomes the list items.
```

[83]:
```
a = "Hello, World!"
print(a.split(",")) # returns ['Hello', ' World!']
```

```
['Hello', ' World!']
```

[ ]:
```
String Concatenation
To concatenate, or combine, two strings you can use the + operator.
```

[84]:
```
a = "Hello"
b = "World"
c = a + b
print(c)
```

```
HelloWorld
```

[85]:
```
#To add a space between them, add a " ":

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

```
Hello World
```

[ ]:
```
String Format
As we learned in the Python Variables chapter, we cannot combine strings and␣
 ↪numbers like this:
```

[ ]:
```
age = 36
txt = "My name is John, I am " + age
```

```
print(txt)
```

[ ]:
```
But we can combine strings and numbers by using the format() method!

The format() method takes the passed arguments, formats them, and places them␣
 ↪in the string where the placeholders {} are:
```

[87]:
```
#Use the format() method to insert numbers into strings:

age = 23
txt = "My name is Aquib, and I am {}"
print(txt.format(age))
```

```
My name is Aquib, and I am 23
```

[ ]:
```
The format() method takes unlimited number of arguments, and are placed into␣
 ↪the respective placeholders
```

[88]:
```
quantity = 3
itemno = 567
price = 49.95
myorder = "I want {} pieces of item {} for {} dollars."
print(myorder.format(quantity, itemno, price))
```

```
I want 3 pieces of item 567 for 49.95 dollars.
```

[ ]:
```
You can use index numbers {0} to be sure the arguments are placed in the␣
 ↪correct placeholders:
```

[89]:
```
quantity = 3
itemno = 567
price = 49.95
myorder = "I want to pay {2} dollars for {0} pieces of item {1}."
print(myorder.format(quantity, itemno, price))
```

```
I want to pay 49.95 dollars for 3 pieces of item 567.
```

[ ]:
```
Escape Character
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:
```

[ ]:
```
You will get an error if you use double quotes inside a string that is␣
 ↪surrounded by double quotes:
```

[93]:
```
txt = "We are the so-called "Vikings" from the north."
print(txt)
```

```
  Cell In[93], line 1
    txt = "We are the so-called "Vikings" from the north."
                                ^
SyntaxError: invalid syntax
```

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

[92]:
```
txt = "We are the so-called \"Vikings\" from the north."
print(txt)
```

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

[ ]:
```
Code        Result
\'        Single Quote
\\        Backslash
\n        New Line
\r        Carriage Return
\t        Tab
\b        Backspace
\f        Form Feed
\ooo        Octal value
\xhh        Hex value
```

[ ]:
```
String Methods
Python has a set of built-in methods that you can use on strings.

Note: All string methods return new values. They do not change the original␣
 ↪string.
```

[ ]:
```
Method                                                  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
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
isnumeric()        Returns True if all characters in the string are numeric
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
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
maketrans()        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
```

5.0.1 Boolean in Python

[ ]:
```
Python Booleans
Booleans represent one of two values: True or False.
```

[2]:
```
print( 4 > 5)
print( 4 < 5)
print( 4 == 5)
```

```
False
True
False
```

[ ]:
```
When you run a condition in an if statement, Python returns True or False:
```

[3]:
```
a = 300
b = 400
if( a > b):
    print(a," is greater than ", b)
else:
    print(b," is greater than ",a)
```

```
400  is greater than  300
```

[ ]:
```
Evaluate Values and Variables
The bool() function allows you to evaluate any value, and give you True or␣
 ↪False in return,
```

[4]:
```
print(bool("Hello! Aquib Sheikh"))
print(bool(15))
```

```
True
True
```

[ ]:
```
Most Values are True
Almost any value is evaluated to True if it has some sort of content.

Any string is True, except empty strings.

Any number is True, except 0.
```

```
Any list, tuple, set, and dictionary are True, except empty ones.
```

[6]:
```
print(bool("Sheikh"))
print(bool(123))
print(bool(["apple","cherry","banana"]))
```

```
True
True
True
```

[ ]:
```
Some Values are False
In fact, there are not many values that evaluate to False, except empty values,
such as (), [], {}, "", the number 0, and the value None. And of course the␣
 ↪value False evaluates to False.
```

[7]:
```
bool(False)
bool(None)
bool(0)
bool("")
bool(())
bool([])
bool({})
```

[7]:
```
False
```

[ ]:
```
Functions can Return a Boolean
You can create functions that returns a Boolean Value:
```

[8]:
```
def myfunction():
    return True
print(myfunction())
```

```
True
```

[ ]:
```
You can execute code based on the Boolean answer of a function:
```

[10]:
```
def myfunction():
    return True
if myfunction():
    print("Yes!")
else:
    print("Not")
```

```
Yes!
```

[ ]:
```
Python also has many built-in functions that return a boolean value,
like the isinstance() function, which can be used to determine if an object is␣
 ↪of a certain data type:
```

[12]:
```
x = 200
print(isinstance(x, int))
```

```
True
```

5.0.2 Python Operators

[ ]:
```
Operators are used to perform operations on variables and values.
```

[ ]:
```
Python divides the operators in the following groups:

Arithmetic operators
Assignment operators
Comparison operators
Logical operators
Identity operators
Membership operators
Bitwise operators
```

[ ]:
```
Python Arithmetic Operators
```

[15]:
```
x = 2
y = 3
print(x+y)
print(x-y)
print(x*y)
print(x/y)
print(x%y)
print(x**y)     # Exponentiation
print(x//y)    # floor division
```

```
5
-1
6
0.6666666666666666
2
8
0
```

[ ]:
```
Python Assignment Operators
```

[ ]:
```
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
```

[ ]:
```
Python Comparison Operators
```

[ ]:
```
Operator                            Name
  ↪        Example
==                                   Equal
  ↪           x == y
!=                                   Not equal
  ↪           x != y
>                                   Greater than
  ↪         x > y
<                                   Less than
  ↪          x < y
>=                                   Greater than or equal to
  ↪          x >= y
<=                                   Less than or equal to
  ↪           x <= y
```

[20]:
```
x = 5
y = 6
print(x == y)
print( x != y)
print(x > y)
print(x < y)
```

```
print(x >= y)
print(x <= y)
```

```
False
True
False
True
False
True
```

[ ]:
```
Python Logical Operators
Logical operators are used to combine conditional statements:
```

[ ]:
```
Operator                                    Description                                 ␣
 ↪                 Example
and                                    Returns True if both 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 the result is␣
 ↪true                       not(x < 5 and x < 10)
```

[21]:
```
x = 5
print( x > 3 and x < 10)
print( x > 3  or x < 10)
print( not( x > 3 and x < 10))
```

```
True
True
False
```

[ ]:
```
Python 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
```

[22]:
```
x = ["Aquib"]
y =["Aiyaz"]
print( x is y)
print( x is not y)
```

```
False
True
```

[ ]:
```
Python 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
```

[24]:
```
x =["Aiyaz","Aquib"]
print("Aiyaz" in x)
print("Sheikh" not in x)
```

```
True
True
```

[ ]:
```
Python Bitwise Operators
Bitwise operators are used to compare (binary) numbers:
```

[ ]:
```
Operator                Name                              Description             ␣
 ↪                             Example
&                         AND                      Sets each bit to 1 if both␣
 ↪bits are 1                              x & y
|                         OR                     Sets each bit to 1 if one of two␣
 ↪bits is 1                      x | y
^                         XOR                      Sets each bit to 1 if only one␣
 ↪of two bits is 1                   x ^ y
~                         NOT                      Inverts all the bits            ␣
 ↪                               ~x
<<              Zero fill left shift        Shift left by pushing zeros in from␣
 ↪the right and let the leftmost bits fall off        x << 2
>>         Signed right shift         Shift right by pushing copies of the␣
 ↪leftmost bit in from the left, and let the rightmost bits fall off
```

[ ]:
```
Operator Precedence
Operator precedence describes the order in which operations are performed.
```

[25]:
```
print((6 + 3) - (6 + 3))
```

```
0
```

[ ]:
```
Multiplication * has higher precedence than addition +, and therefor␣
 ↪multiplications are evaluated before additions:
```

[26]:
```
print(100 + 5 * 3)
```

```
115
```

[ ]:
```
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
```

[ ]:
```
If two operators have the same precedence, the expression is evaluated from␣
 ↪left to right.
```

[27]:
```
print(5 + 4 - 7 + 3)
```

```
5
```

5.0.3 Python Lists

[ ]:
```
List
Lists are used to store multiple items in a single variable.
List Items
List items are ordered, changeable, and allow duplicate values.

List items are indexed, the first item has index [0], the second item has index␣
 ↪[1] etc.

Ordered
When we say that lists are ordered, it means that the items have a defined␣
 ↪order, and that order will not change.
```

```
If you add new items to a list, the new items will be placed at the end of the␣
 ↪list.

Note: There are some list methods that will change the order, but in general:␣
 ↪the order of the items will not change.

Changeable
The list is changeable, meaning that we can change, add, and remove items in a␣
 ↪list after it has been created.

Allow Duplicates
Since lists are indexed, lists can have items with the same value:
```

[28]:
```
thislist = ["apple","banana","cherry","apple"]
print(thislist)
```

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

[ ]:
```
List Length
To determine how many items a list has, use the len() function:
```

[29]:
```
thislist = ["apple", "banana", "cherry"]
print(len(thislist))
```

```
3
```

[ ]:
```
List Items - Data Types
List items can be of any data type:
```

[30]:
```
list1 = ["apple", "banana", "cherry"]
list2 = [1, 5, 7, 9, 3]
list3 = [True, False, False]
```

[ ]:
```
A list can contain different data types:
```

[31]:
```
list1 = ["abc", 34, True, 40, "male"]
```

[32]:
```
print(list1)
```

```
['abc', 34, True, 40, 'male']
```

[ ]:
```
type()
From Python's perspective, lists are defined as objects with the data type␣
 ↪'list':
```

[33]:
```
mylist = ["apple", "banana", "cherry"]
print(type(mylist))
```

```
<class 'list'>
```

[ ]:
```
The list() Constructor
It is also possible to use the list() constructor when creating a new list.
```

[34]:
```
thislist = list(("apple", "banana", "cherry")) # note the double round-brackets
print(thislist)
```

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

[ ]:
```
Python Collections (Arrays)
There are four collection data types in the Python programming language:

List is a collection which is ordered and changeable. Allows duplicate members.
Tuple is a collection which is ordered and unchangeable. Allows duplicate␣
 ↪members.
Set is a collection which is unordered, unchangeable*, and unindexed. No␣
 ↪duplicate members.
Dictionary is a collection which is ordered** and changeable. No duplicate␣
 ↪members.
```

[ ]:
```
Access Items
List items are indexed and you can access them by referring to the index number:
```

[35]:
```
thislist = ["apple", "banana", "cherry"]
print(thislist[1])
```

```
banana
```

[ ]:
```
Negative Indexing
Negative indexing means start from the end

-1 refers to the last item, -2 refers to the second last item etc.
```

[36]:
```
thislist = ["apple", "banana", "cherry"]
print(thislist[-1])
```

```
cherry
```

[ ]:
```
Range of Indexes
You can specify a range of indexes by specifying where to start and where to␣
 ↪end the range.

When specifying a range, the return value will be a new list with the specified␣
 ↪items.
```

[37]:
```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(thislist[2:5])
```

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

[ ]:
```
Note: The search will start at index 2 (included) and end at index 5 (not␣
 ↪included).

Remember that the first item has index 0.
```

[38]:
```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(thislist[:4])
```

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

[39]:
```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(thislist[2:])
```

```
['cherry', 'orange', 'kiwi', 'melon', 'mango']
```

[ ]:
```
Range of Negative Indexes
Specify negative indexes if you want to start the search from the end of the␣
 ↪list:
```

[40]:
```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(thislist[-4:-1])
```

```
['orange', 'kiwi', 'melon']
```

[ ]:
```
Check if Item Exists
To determine if a specified item is present in a list use the in keyword:
```

[41]:
```
thislist = ["apple", "banana", "cherry"]
if "apple" in thislist:
  print("Yes, 'apple' is in the fruits list")
```

```
Yes, 'apple' is in the fruits list
```

[ ]:
```
Change Item Value
To change the value of a specific item, refer to the index number:
```

[42]:
```
thislist = ["apple", "banana", "cherry"]
thislist[1] = "blackcurrant"
print(thislist)
```

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

[ ]:
```
Change a Range of Item Values
To change the value of items within a specific range, define a list with the␣
 ↪new values,
and refer to the range of index numbers where you want to insert the new values:
```

[43]:
```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "mango"]
thislist[1:3] = ["blackcurrant", "watermelon"]
print(thislist)
```

```
['apple', 'blackcurrant', 'watermelon', 'orange', 'kiwi', 'mango']
```

[ ]:
```
If you insert more items than you replace, the new items will be inserted where␣
 ↪you specified,
and the remaining items will move accordingly:
```

[44]:
```
thislist = ["apple", "banana", "cherry"]
thislist[1:2] = ["blackcurrant", "watermelon"]
print(thislist)
```

```
['apple', 'blackcurrant', 'watermelon', 'cherry']
```

[ ]:
```
If you insert less items than you replace, the new items will be inserted where␣
 ↪you specified,
and the remaining items will move accordingly:
```

[45]:
```
thislist = ["apple", "banana", "cherry"]
thislist[1:3] = ["watermelon"]
print(thislist)
```

```
['apple', 'watermelon']
```

[ ]:
```
Insert Items
To insert a new list item, without replacing any of the existing values, we can␣
 ↪use the insert() method.

The insert() method inserts an item at the specified index:
```

[46]:
```
thislist = ["apple", "banana", "cherry"]
thislist.insert(2, "watermelon")
print(thislist)
```

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

[ ]:
```
Append Items
To add an item to the end of the list, use the append() method:
```

[47]:
```
thislist = ["apple", "banana", "cherry"]
thislist.append("orange")
print(thislist)
```

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

[ ]:
```
Insert Items
To insert a list item at a specified index, use the insert() method.

The insert() method inserts an item at the specified index:
```

[48]:
```
thislist = ["apple", "banana", "cherry"]
thislist.insert(1, "orange")
print(thislist)
```

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

[ ]:
```
Extend List
To append elements from another list to the current list, use the extend()␣
↪method.
```

[49]:
```
thislist = ["apple", "banana", "cherry"]
tropical = ["mango", "pineapple", "papaya"]
thislist.extend(tropical)
print(thislist)
```

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

[ ]:
```
Add Any Iterable
The extend() method does not have to append lists, you can add any iterable␣
↪object (tuples, sets, dictionaries etc.).
```

[50]:
```
thislist = ["apple", "banana", "cherry"]
thistuple = ("kiwi", "orange")
thislist.extend(thistuple)
print(thislist)
```

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

[ ]:
```
Remove Specified Item
The remove() method removes the specified item.
```

[51]:
```
thislist = ["apple", "banana", "cherry"]
thislist.remove("banana")
print(thislist)
```

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

[ ]:
```
Remove Specified Index
The pop() method removes the specified index.
```

[52]:
```
thislist = ["apple", "banana", "cherry"]
thislist.pop(1)
print(thislist)
```

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

[ ]:
```
If you do not specify the index, the pop() method removes the last item.
```

[53]:
```
thislist = ["apple", "banana", "cherry"]
thislist.pop()
```

```
print(thislist)
```

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

[ ]:
```
The del keyword also removes the specified index:
```

[54]:
```
thislist = ["apple", "banana", "cherry"]
del thislist[0]
print(thislist)
```

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

[ ]:
```
The del keyword can also delete the list completely.
```

[55]:
```
thislist = ["apple", "banana", "cherry"]
del thislist
```

[ ]:
```
Clear the List
The clear() method empties the list.

The list still remains, but it has no content.
```

[56]:
```
thislist = ["apple", "banana", "cherry"]
thislist.clear()
print(thislist)
```

```
[]
```

[ ]:
```
Loop Through a List
You can loop through the list items by using a for loop:
```

[57]:
```
thislist = ["apple", "banana", "cherry"]
for x in thislist:
  print(x)
```

```
apple
banana
cherry
```

[ ]:
```
Loop Through the Index Numbers
You can also loop through the list items by referring to their index number.

Use the range() and len() functions to create a suitable iterable.
```

[58]:
```
thislist = ["apple", "banana", "cherry"]
for i in range(len(thislist)):
  print(thislist[i])
```

```
apple
banana
```

```
cherry
```

[ ]:
```
Using a While Loop
You can loop through the list items by using a while loop.

Use the len() function to determine the length of the list,
then start at 0 and loop your way through the list items by referring to their␣
 ↪indexes.

Remember to increase the index by 1 after each iteration.
```

[59]:
```
thislist = ["apple", "banana", "cherry"]
i = 0
while i < len(thislist):
  print(thislist[i])
  i = i + 1
```

```
apple
banana
cherry
```

[ ]:
```
Looping Using List Comprehension
List Comprehension offers the shortest syntax for looping through lists:
```

[1]:
```
thislist = ["apple", "banana", "cherry"]
[print(x) for x in thislist]
```

```
apple
banana
cherry
```

[1]:
```
[None, None, None]
```

[ ]:
```
With list comprehension you can do all that with only one line of code:
```

[2]:
```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]

newlist = [x for x in fruits if "a" in x]

print(newlist)
```

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

[ ]:
```
Sort List Alphanumerically
List objects have a sort() method that will sort the list alphanumerically,␣
 ↪ascending, by default:
```

[1]:
```
thislist = ["mango","apple","orange","kiwi","pineapple","banana"]
thislist.sort()
```

```
print(thislist)
```

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

[2]:
```
thislist = [30,12,4,-1,8,8,9,3,5]
thislist.sort()
print(thislist)
```

```
[-1, 3, 4, 5, 8, 8, 9, 12, 30]
```

[ ]:
```
Sort Descending
To sort descending, use the keyword argument reverse = True:
```

[3]:
```
thislist = ["mango","apple","orange","kiwi","pineapple","banana"]
thislist.sort( reverse = True)
print(thislist)
```

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

[4]:
```
thislist = [30,12,4,-1,8,8,9,3,5]
thislist.sort(reverse = True)
print(thislist)
```

```
[30, 12, 9, 8, 8, 5, 4, 3, -1]
```

[ ]:
```
Customize Sort Function
You can also customize your own function by using the keyword argument key =␣
 ↪function.

The function will return a number that will be used to sort the list (the␣
 ↪lowest number first):
```

[5]:
```
#Sort the list based on how close the number is to 50:

def myfunc(n):
  return abs(n - 50)

thislist = [100, 50, 65, 82, 23]
thislist.sort(key = myfunc)
print(thislist)
```

```
[50, 65, 23, 82, 100]
```

[ ]:
```
Case Insensitive Sort
By default the sort() method is case sensitive, resulting in all capital␣
 ↪letters being sorted before lower case letters:
```

[6]:
```
#Case sensitive sorting can give an unexpected result:

thislist = ["banana", "Orange", "Kiwi", "cherry"]
```

```
thislist.sort()
print(thislist)
```

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

[ ]:
```
Luckily we can use built-in functions as key functions when sorting a list.

So if you want a case-insensitive sort function, use str.lower as a key␣
 ↪function:
```

[7]:
```
#Perform a case-insensitive sort of the list:

thislist = ["banana", "Orange", "Kiwi", "cherry"]
thislist.sort(key = str.lower)
print(thislist)
```

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

[ ]:
```
Reverse Order
What if you want to reverse the order of a list, regardless of the alphabet?

The reverse() method reverses the current sorting order of the elements.
```

[9]:
```
thislist = ["banana","Orange","KIWI","cherry"]
thislist.reverse()
print(thislist)
```

```
['cherry', 'KIWI', 'Orange', 'banana']
```

[ ]:
```
Copy a List
You cannot copy a list simply by typing list2 = list1,
because: list2 will only be a reference to list1, and changes made in list1␣
 ↪will automatically also be made in list2.

There are ways to make a copy, one way is to use the built-in List method␣
 ↪copy().
```

[10]:
```
thislist = ["aquib","aiyaz","unknown","known"]
mylist = thislist.copy()
print(mylist)
```

```
['aquib', 'aiyaz', 'unknown', 'known']
```

[ ]:
```
Another way to make a copy is to use the built-in method list().
```

[11]:
```
thislist = ["apple", "banana", "cherry"]
mylist = list(thislist)
print(mylist)
```

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

[ ]:
```
Join Two Lists
There are several ways to join, or concatenate, two or more lists in Python.

One of the easiest ways are by using the + operator.
```

[13]:
```
list1 = [ "a","q","u","i","b"]
list2 = [1,2,3,4,5]

list3 = list1 + list2
print(list3)
```

```
['a', 'q', 'u', 'i', 'b', 1, 2, 3, 4, 5]
```

[ ]:
```
Another way to join two lists is by appending all the items from list2 into␣
 ↪list1, one by one:
```

[14]:
```
list1 = ["a", "b" , "c"]
list2 = [1, 2, 3]

for x in list2:
  list1.append(x)

print(list1)
```

```
['a', 'b', 'c', 1, 2, 3]
```

[ ]:
```
Or you can use the extend() method, where the purpose is to add elements from␣
 ↪one list to another list:
```

[15]:
```
list1 = ["a", "b" , "c"]
list2 = [1, 2, 3]

list1.extend(list2)
print(list1)
```

```
['a', 'b', 'c', 1, 2, 3]
```

[ ]:
```
List Methods
Python has a set of built-in methods that you can use on lists.

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 item with the specified␣
 ↪value
reverse()                                   Reverses the order of the list
sort()                                       Sorts the list
```

5.0.4 Python Tuples

[ ]:
```
Tuples are used to store multiple items in a single variable.
A tuple is a collection which is ordered and unchangeable.

Tuples are written with round brackets.
```

[16]:
```
thistuple = ("apple", "banana", "cherry")
print(thistuple)
```

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

[ ]:
```
Tuple Items
Tuple items are ordered, unchangeable, and allow duplicate values.

Tuple items are indexed, the first item has index [0], the second item has␣
 ↪index [1] etc.
```

[17]:
```
thistuple = ("apple", "banana", "cherry", "apple", "cherry")
print(thistuple)
```

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

[ ]:
```
Tuple Length
To determine how many items a tuple has, use the len() function:
```

[18]:
```
thistuple = ("apple", "banana", "cherry")
print(len(thistuple))
```

```
3
```

[ ]:
```
Create Tuple With One Item
To create a tuple with only one item, you have to add a comma after the item
otherwise Python will not recognize it as a tuple.
```

[19]:
```
thistuple = ("apple",)
print(type(thistuple))

#NOT a tuple
thistuple = ("apple")
print(type(thistuple))
```

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

[ ]:
```
Tuple Items - Data Types
Tuple items can be of any data type:
```

[20]:
```
tuple1 = ("apple", "banana", "cherry")
tuple2 = (1, 5, 7, 9, 3)
tuple3 = (True, False, False)
```

[ ]:
```
A tuple can contain different data types:
```

[21]:
```
tuple1 = ("abc", 34, True, 40, "male")
```

[22]:
```
print(tuple1)
```

```
('abc', 34, True, 40, 'male')
```

[ ]:
```
The tuple() Constructor
It is also possible to use the tuple() constructor to make a tuple.
```

[23]:
```
#Using the tuple() method to make a tuple:

thistuple = tuple(("apple", "banana", "cherry")) # note the double␣
 ↪round-brackets
print(thistuple)
```

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

[ ]:
```
Python Collections (Arrays)
There are four collection data types in the Python programming language:

List ->       It is a collection which is ordered and changeable. Allows␣
 ↪duplicate members.
Tuple ->      It is a collection which is ordered and unchangeable. Allows␣
 ↪duplicate members.
Set  ->       It is a collection which is unordered, unchangeable*, and␣
 ↪unindexed. No duplicate members.
Dictionary -> It is a collection which is ordered** and changeable. No␣
 ↪duplicate members.
```

[ ]:
```
Access Tuple Items
You can access tuple items by referring to the index number, inside square␣
 ↪brackets:
```

[25]:
```
thistuple = ("apple", "banana", "cherry")
print(thistuple[1])
```

```
banana
```

[ ]:
```
Negative Indexing
Negative indexing means start from the end.

-1 refers to the last item, -2 refers to the second last item etc.
```

[26]:
```
thistuple = ("apple", "banana", "cherry")
print(thistuple[-1])
```

```
cherry
```

[ ]:
```
Range of Indexes
You can specify a range of indexes by specifying where to start and where to␣
 ↪end the range.

When specifying a range, the return value will be a new tuple with the␣
 ↪specified items.
```

[27]:
```
thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
print(thistuple[2:5])
```

```
('cherry', 'orange', 'kiwi')
```

[ ]:
```
Note: The search will start at index 2 (included) and end at index 5 (not␣
 ↪included).

Remember that the first item has index 0.
```

[28]:
```
# This example returns the items from the beginning to, but NOT included,␣
 ↪"kiwi":

thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
print(thistuple[:4])
```

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

[29]:
```
# This example returns the items from "cherry" and to the end:

thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
print(thistuple[2:])
```

```
('cherry', 'orange', 'kiwi', 'melon', 'mango')
```

[ ]:
```
Range of Negative Indexes
Specify negative indexes if you want to start the search from the end of the␣
 ↪tuple:
```

[30]:
```
thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
print(thistuple[-4:-1])
```

```
('orange', 'kiwi', 'melon')
```

[ ]:
```
Check if Item Exists
To determine if a specified item is present in a tuple use the in keyword:
```

[31]:
```
thistuple = ("apple", "banana", "cherry")
if "apple" in thistuple:
  print("Yes, 'apple' is in the fruits tuple")
```

```
Yes, 'apple' is in the fruits tuple
```

[ ]:
```
Tuples are unchangeable, meaning that you cannot change, add, or remove items␣
 ↪once the tuple is created.

But there are some workarounds.

Change Tuple Values
Once a tuple is created, you cannot change its values. Tuples are unchangeable,␣
 ↪or immutable as it also is called.

But there is a workaround. You can convert the tuple into a list, change the␣
 ↪list, and convert the list back into a tuple.
```

[1]:
```
#Convert the tuple into a list to be able to change it:

x = ("apple", "banana", "cherry")
y = list(x)
y[1] = "kiwi"
x = tuple(y)

print(x)
```

```
('apple', 'kiwi', 'cherry')
```

[ ]:
```
Add Items
Since tuples are immutable, they do not have a built-in append() method, but␣
 ↪there are other ways to add items to a tuple.

1. Convert into a list: Just like the workaround for changing a tuple, you can␣
 ↪convert it into a list, add your item(s),
and convert it back into a tuple.
```

[2]:
```
thistuple = ("apple", "banana", "cherry")
y = list(thistuple)
y.append("orange")
thistuple = tuple(y)
```

[3]:
```
print(thistuple)
```

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

[ ]:
```
2. Add tuple to a tuple. You are allowed to add tuples to tuples, so if you␣
 ↪want to add one item, (or many),
create a new tuple with the item(s),
and add it to the existing tuple:
```

[4]:
```
thistuple = ("apple", "banana", "cherry")
y = ("orange",)
thistuple += y

print(thistuple)
```

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

[ ]:
```
Remove Items
Note: You cannot remove items in a tuple.

Tuples are unchangeable, so you cannot remove items from it,
but you can use the same workaround as we used for changing and adding tuple␣
 ↪items:
```

[7]:
```
thistuple = ("apple","banana","guava")
y = list(thistuple)
y.remove("apple")
thistuple = tuple(y)
print(thistuple)
```

```
('banana', 'guava')
```

[ ]:
```
Or you can delete the tuple completely using del keyword
```

[9]:
```
thistuple = ("apple","banana","guava")
del thistuple
print(thistuple)   # this will raise an error
```

[ ]:
```
Unpacking a Tuple
When we create a tuple, we normally assign values to it. This is called␣
 ↪"packing" a tuple:
```

[11]:
```
packing = ("apple","banana","guava")
print(packing)
```

```
(red, green, yellow) = packing
print(red)
print(green)
print(yellow)
```

```
('apple', 'banana', 'guava')
apple
banana
guava
```

[ ]:
```
Note: The number of variables must match the number of values in the tuple, if␣
 ↪not,
you must use an asterisk to collect the remaining values as a list.
```

[ ]:
```
Using Asterisk*
If the number of variables is less than the number of values,
you can add an * to the variable name and the values will be assigned to the␣
 ↪variable as a list:
```

[12]:
```
fruits = ("apple", "banana", "cherry", "strawberry", "raspberry")

(green, yellow, *red) = fruits

print(green)
print(yellow)
print(red)
```

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

[ ]:
```
If the asterisk is added to another variable name than the last,
Python will assign values to the variable until the number of values left␣
 ↪matches the number of variables left.
```

[13]:
```
fruits = ("apple", "mango", "papaya", "pineapple", "cherry")

(green, *tropic, red) = fruits

print(green)
print(tropic)
print(red)
```

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

[ ]:
```
Loop Through a Tuple
You can loop through the tuple items by using a for loop.
```

[14]:
```
thistuple = ("apple", "banana", "cherry")
for x in thistuple:
  print(x)
```

```
apple
banana
cherry
```

[ ]:
```
Loop Through the Index Numbers
You can also loop through the tuple items by referring to their index number.

Use the range() and len() functions to create a suitable iterable.
```

[15]:
```
thistuple = ("apple", "banana", "cherry")
for i in range(len(thistuple)):
  print(thistuple[i])
```

```
apple
banana
cherry
```

[ ]:
```
Using a While Loop
You can loop through the tuple items by using a while loop.

Use the len() function to determine the length of the tuple, then start at 0 and
loop your way through the tuple items by referring to their indexes.

Remember to increase the index by 1 after each iteration.
```

[16]:
```
thistuple = ("apple", "banana", "cherry")
i = 0
while i < len(thistuple):
  print(thistuple[i])
  i = i + 1
```

```
apple
banana
cherry
```

[ ]:
```
Join Two Tuples
To join two or more tuples you can use the + operator:
```

[17]:
```
tuple1 = ("a", "b" , "c")
tuple2 = (1, 2, 3)

tuple3 = tuple1 + tuple2
```

```
print(tuple3)
```

```
('a', 'b', 'c', 1, 2, 3)
```

[ ]:
```
Multiply Tuples
If you want to multiply the content of a tuple a given number of times, you can␣
 ↪use the * operator:
```

[18]:
```
fruits = ("apple", "banana", "cherry")
mytuple = fruits * 2

print(mytuple)
```

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

[ ]:
```
Tuple Methods
Python has two built-in methods that you can use on tuples.

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
```

5.0.5 Set In Python

[ ]:
```
Sets are used to store multiple items in a single variable.
A set is a collection which is unordered, unchangeable*, and unindexed.

* Note: Set items are unchangeable, but you can remove items and add new items.

Sets are written with curly brackets.
```

[19]:
```
thisset = {"apple", "banana", "cherry"}           # Creating a set
print(thisset)
```

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

[ ]:
```
Note: Sets are unordered, so you cannot be sure in which order the items will␣
 ↪appear.
```

[ ]:
```
Set Items
Set items are unordered, unchangeable, and do not allow duplicate values.

Unordered
Unordered means that the items in a set do not have a defined order.

Set items can appear in a different order every time you use them, and cannot␣
 ↪be referred to by index or key.
```

```
Unchangeable
Set items are unchangeable, meaning that we cannot change the items after the␣
 ↪set has been created.

Once a set is created, you cannot change its items, but you can remove items␣
 ↪and add new items.

Duplicates Not Allowed
Sets cannot have two items with the same value.
```

[20]:
```
thisset = {"apple", "banana", "cherry", "apple"}

print(thisset)
```

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

[ ]:
```
Note: The values True and 1 are considered the same value in sets, and are␣
 ↪treated as duplicates:
```

[21]:
```
thisset = {"apple", "banana", "cherry", True, 1, 2}

print(thisset)
```

```
{True, 2, 'apple', 'banana', 'cherry'}
```

[ ]:
```
Get the Length of a Set
To determine how many items a set has, use the len() function.
```

[22]:
```
thisset = {"apple", "banana", "cherry"}

print(len(thisset))
```

```
3
```

[ ]:
```
Set Items - Data Types
Set items can be of any data type:
```

[ ]:
```
set1 = {"apple", "banana", "cherry"}
set2 = {1, 5, 7, 9, 3}
set3 = {True, False, False}
```

[ ]:
```
A set can contain different data types:
```

[ ]:
```
set1 = {"abc", 34, True, 40, "male"}
```

[ ]:
```
type()
From Python's perspective, sets are defined as objects with the data type 'set':
```

[23]:
```
myset = {"apple", "banana", "cherry"}
print(type(myset))
```

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

[ ]:
```
The set() Constructor
It is also possible to use the set() constructor to make a set.
```

[24]:
```
thisset = set(("apple", "banana", "cherry")) # note the double round-brackets
print(thisset)
```

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

[ ]:
```
Access Items
You cannot access items in a set by referring to an index or a key.

But you can loop through the set items using a for loop, or ask if a specified␣
 ↪value is present in a set, by using the in keyword.
```

[25]:
```
thisset = {"apple", "banana", "cherry"}

for x in thisset:
  print(x)
```

```
apple
banana
cherry
```

[26]:
```
#Check if "banana" is present in the set:

thisset = {"apple", "banana", "cherry"}

print("banana" in thisset)
```

```
True
```

[ ]:
```
Change Items
Once a set is created, you cannot change its items, but you can add new items.
```

[27]:
```
# Add an item to a set, using the add() method:

thisset = {"apple", "banana", "cherry"}

thisset.add("orange")

print(thisset)
```

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

[ ]:
```
Add Sets
To add items from another set into the current set, use the update() method.
```

[28]:
```
# Add elements from tropical into thisset:

thisset = {"apple", "banana", "cherry"}
tropical = {"pineapple", "mango", "papaya"}

thisset.update(tropical)

print(thisset)
```

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

[ ]:
```
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.).
```

[29]:
```
# Add elements of a list to at set:

thisset = {"apple", "banana", "cherry"}
mylist = ["kiwi", "orange"]

thisset.update(mylist)

print(thisset)
```

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

[ ]:
```
Remove Item
To remove an item in a set, use the remove(), or the discard() method.
```

[30]:
```
# Remove "banana" by using the remove() method:

thisset = {"apple", "banana", "cherry"}

thisset.remove("banana")

print(thisset)
```

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

[31]:
```
# Remove "banana" by using the discard() method:

thisset = {"apple", "banana", "cherry"}

thisset.discard("banana")

print(thisset)
```

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

[ ]:
```
Note: If the item to remove does not exist, discard() will NOT raise an error.
```

[ ]:
```
You can also use the pop() method to remove an item, but this method will␣
  ↪remove a random item,
so you cannot be sure what item that gets removed.

The return value of the pop() method is the removed item.
```

[32]:
```
# Remove a random item by using the pop() method:

thisset = {"apple", "banana", "cherry"}

x = thisset.pop()

print(x)

print(thisset)
```

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

[ ]:
```
Note: Sets are unordered, so when using the pop() method, you do not know which␣
  ↪item that gets removed.
```

[33]:
```
# The clear() method empties the set:

thisset = {"apple", "banana", "cherry"}

thisset.clear()

print(thisset)
```

```
set()
```

[34]:
```
# The del keyword will delete the set completely:

thisset = {"apple", "banana", "cherry"}

del thisset

print(thisset)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[34], line 7
      3 thisset = {"apple", "banana", "cherry"}
      5 del thisset
```

```
----> 7 print(thisset)

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

[ ]:
```
Loop Items
You can loop through the set items by using a for loop:
```

[35]:
```
thisset = {"apple", "banana", "cherry"}

for x in thisset:
  print(x)
```

```
apple
banana
cherry
```

[ ]:
```
Join Two Sets
There are several ways to join two or more sets in Python.

You can use the union() method that returns a new set containing all items from␣
 ↪both sets,
    or the update() method that inserts all the items from one set into another:
```

[36]:
```
#The union() method returns a new set with all items from both sets:

set1 = {"a", "b" , "c"}
set2 = {1, 2, 3}

set3 = set1.union(set2)
print(set3)
```

```
{1, 2, 3, 'c', 'a', 'b'}
```

[37]:
```
#The update() method inserts the items in set2 into set1:

set1 = {"a", "b" , "c"}
set2 = {1, 2, 3}

set1.update(set2)
print(set1)
```

```
{1, 2, 3, 'c', 'a', 'b'}
```

[ ]:
```
Note: Both union() and update() will exclude any duplicate items.
```

[ ]:
```
Keep ONLY the Duplicates
The intersection_update() method will keep only the items that are present in␣
 ↪both sets.
```

[38]:
```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}

x.intersection_update(y)

print(x)
```

```
{'apple'}
```

[ ]:
```
The intersection() method will return a new set, that only contains the items␣
 ↪that are present in both sets.
```

[39]:
```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}

z = x.intersection(y)

print(z)
```

```
{'apple'}
```

[ ]:
```
Keep All, But NOT the Duplicates
The symmetric_difference_update() method will keep only the elements that are␣
 ↪NOT present in both sets.
```

[40]:
```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}

x.symmetric_difference_update(y)

print(x)
```

```
{'google', 'banana', 'microsoft', 'cherry'}
```

[ ]:
```
The symmetric_difference() method will return a new set, that contains only the␣
 ↪elements that are NOT present in both sets.
```

[41]:
```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}

z = x.symmetric_difference(y)

print(z)
```

```
{'google', 'cherry', 'banana', 'microsoft'}
```

[ ]:
```
Note: The values True and 1 are considered the same value in sets, and are␣
 ↪treated as duplicates:
```

[42]:
```
#True and 1 is considered the same value:

x = {"apple", "banana", "cherry", True}
y = {"google", 1, "apple", 2}

z = x.symmetric_difference(y)
```

[ ]:
```
Set Methods
Python has a set of built-in methods that you can use on sets.

Method                                                  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
issuperset()                                             Returns whether this set␣
 ↪contains another set or not
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
```

5.0.6 Python Dictionary

[ ]:
```
Dictionaries are used to store data values in key:value pairs.
```

```
A dictionary is a collection which is ordered*, changeable and do not allow␣
 ↪duplicates.
Dictionaries are written with curly brackets, and have keys and values:
```

[1]:
```
thisdict = {
    "brand": "ford",
    "model": "Mustang",
    "year": 1964
}
print(thisdict)
```

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

[ ]:
```
Dictionary Items
Dictionary items are ordered, changeable, and does not allow duplicates.

Dictionary items are presented in key:value pairs, and can be referred to by␣
 ↪using the key name.
```

[5]:
```
thisdict = {
    "brand": "ford",
    "model": "Mustang",
    "year": 1964
}
print(thisdict["model"])
```

```
Mustang
```

[ ]:
```
Duplicates Not Allowed
Dictionaries cannot have two items with the same key:
```

[6]:
```
thisdict = {
    "brand": "ford",
    "model": "Mustang",
    "year": 1964,
    "year" : 2020
}
print(thisdict)
```

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

[ ]:
```
Dictionary Length
To determine how many items a dictionary has, use the len() function:
```

[7]:
```
print(len(thisdict))
```

```
3
```

[ ]:
```
Dictionary Items - Data Types
The values in dictionary items can be of any data type:
```

[8]:
```
thisdict = {
    "brand" : "ford",
    "model" : "Mustang",
    "year" : 1964,
    "colors": ["red","white","blue"],
    "electric": False
}
print(thisdict)
```

```
{'brand': 'ford', 'model': 'Mustang', 'year': 1964, 'colors': ['red', 'white',
'blue'], 'electric': False}
```

[ ]:
```
type()
From Python's perspective, dictionaries are defined as objects with the data␣
 ↪type 'dict':
```

[9]:
```
print(type(thisdict))
```

```
<class 'dict'>
```

[ ]:
```
The dict() Constructor
It is also possible to use the dict() constructor to make a dictionary.
```

[10]:
```
thisdict = dict(name = "Aquib Sheikh", age = 23, country = "India")
print(thisdict)
```

```
{'name': 'Aquib Sheikh', 'age': 23, 'country': 'India'}
```

[ ]:
```
Accessing Items
You can access the items of a dictionary by referring to its key name, inside␣
 ↪square brackets:
```

[12]:
```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
x = thisdict["model"]
print(x)
```

```
Mustang
```

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

[13]:
```
x = thisdict.get("model")
print(x)
```

```
Mustang
```

[ ]:
```
Get Keys
The keys() method will return a list of all the keys in the dictionary.
```

[14]:
```
x = thisdict.keys()
print(x)
```

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

[15]:
```
#Add a new item to the original dictionary, and see that the keys list gets␣
 ↪updated as well:

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

x = car.keys()

print(x) #before the change

car["color"] = "white"

print(x) #after the change
```

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

[ ]:
```
Get Values
The values() method will return a list of all the values in the dictionary.
```

[16]:
```
x = thisdict.values()
print(x)
```

```
dict_values(['Ford', 'Mustang', 1964])
```

[17]:
```
#Make a change in the original dictionary, and see that the values list gets␣
 ↪updated as well:

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

x = car.values()
```

```
print(x) #before the change

car["year"] = 2020

print(x) #after the change
```

```
dict_values(['Ford', 'Mustang', 1964])
dict_values(['Ford', 'Mustang', 2020])
```

[18]:
```
#Add a new item to the original dictionary, and see that the values list gets␣
 ↪updated as well:

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

x = car.values()

print(x) #before the change

car["color"] = "red"

print(x) #after the change
```

```
dict_values(['Ford', 'Mustang', 1964])
dict_values(['Ford', 'Mustang', 1964, 'red'])
```

[ ]:
```
Get Items
The items() method will return each item in a dictionary, as tuples in a list.
```

[19]:
```
x = thisdict.items()
print(x)
```

```
dict_items([('brand', 'Ford'), ('model', 'Mustang'), ('year', 1964)])
```

[20]:
```
#Make a change in the original dictionary, and see that the items list gets␣
 ↪updated as well:

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

x = car.items()
```

```
print(x) #before the change

car["year"] = 2020

print(x) #after the change
```

```
dict_items([('brand', 'Ford'), ('model', 'Mustang'), ('year', 1964)])
dict_items([('brand', 'Ford'), ('model', 'Mustang'), ('year', 2020)])
```

[21]:
```
#Add a new item to the original dictionary, and see that the items list gets␣
 ↪updated as well:

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

x = car.items()

print(x) #before the change

car["color"] = "red"

print(x) #after the change
```

```
dict_items([('brand', 'Ford'), ('model', 'Mustang'), ('year', 1964)])
dict_items([('brand', 'Ford'), ('model', 'Mustang'), ('year', 1964), ('color',
'red')])
```

[ ]:
```
Check if Key Exists
To determine if a specified key is present in a dictionary use the in keyword:
```

[22]:
```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
if "model" in thisdict:
  print("Yes, 'model' is one of the keys in the thisdict dictionary")
```

```
Yes, 'model' is one of the keys in the thisdict dictionary
```

[ ]:
```
Change Values
You can change the value of a specific item by referring to its key name:
```

[24]:
```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
```

```
  "year": 1964
}
thisdict["year"] = 2018
print(thisdict)
```

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

[ ]:
```
Update Dictionary
The update() method will update the dictionary with the items from the given␣
 ↪argument.

The argument must be a dictionary, or an iterable object with key:value pairs.
```

[26]:
```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.update({"year": 2020})
print(
    thisdict)
```

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

[ ]:
```
Adding Items
Adding an item to the dictionary is done by using a new index key and assigning␣
 ↪a value to it:
```

[27]:
```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict["color"] = "red"
print(thisdict)
```

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

[ ]:
```
Update Dictionary
The update() method will update the dictionary with the items from a given␣
 ↪argument.
If the item does not exist, the item will be added.

The argument must be a dictionary, or an iterable object with key:value pairs.
```

[29]:
```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
```

```
}
thisdict.update({"color": "red"})

print(thisdict)
```

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

[ ]:
```
Removing Items
There are several methods to remove items from a dictionary:
```

[30]:
```
# The pop() method removes the item with the specified key name:

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

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

[31]:
```
# The 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)
```

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

[32]:
```
# The del keyword removes the item with the specified key name:

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

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

[33]:
```
# 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.
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[33], line 9
      3 thisdict = {
      4   "brand": "Ford",
      5   "model": "Mustang",
      6   "year": 1964
      7 }
      8 del thisdict
----> 9 print(thisdict) #this will cause an error because "thisdict" no longer␣
 ↪exists.

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

[34]:
```
# The clear() method empties the dictionary:

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

```
{}
```

[ ]:
```
Loop Through a Dictionary
You can loop through a dictionary by using a for loop.

When looping through a dictionary, the return value are the keys of the␣
 ↪dictionary,
but there are methods to return the values as well.
```

[39]:
```
thisdict =        {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
```

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

```
brand
model
year
```

[38]:
```
# Print all values in the dictionary, one by one:

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

```
Ford
Mustang
1964
```

[37]:
```
thisdict =        {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
for x in thisdict.values():
  print(x)
```

```
Ford
Mustang
1964
```

[41]:
```
# You can use the keys() method to return the keys of a dictionary:

thisdict =        {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
for x in thisdict.keys():
  print(x)
```

```
brand
model
year
```

[ ]:
```
Loop through both keys and values, by using the items() method:
```

[42]:
```
thisdict =        {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
for x, y in thisdict.items():
  print(x, y)
```

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

[ ]:
```
Copy a Dictionary
You cannot copy a dictionary simply by typing dict2 = dict1,
because: dict2 will only be a reference to dict1, and changes made in dict1␣
 ↪will automatically also be made in dict2.

There are ways to make a copy, one way is to use the built-in Dictionary method␣
 ↪copy().
```

[43]:
```
#Make a copy of a dictionary with the copy() method:

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

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

[ ]:
```
Another way to make a copy is to use the built-in function dict().
```

[44]:
```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
mydict = dict(thisdict)
print(mydict)
```

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

[ ]:
```
Nested Dictionaries
A dictionary can contain dictionaries, this is called nested dictionaries.
```

[46]:
```
# Create a dictionary that contain three dictionaries:

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

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

[ ]:
```
Or, if you want to add three dictionaries into a new dictionary:
```

[47]:
```
child1 = {
  "name" : "Emil",
  "year" : 2004
}
child2 = {
  "name" : "Tobias",
  "year" : 2007
}
child3 = {
  "name" : "Linus",
  "year" : 2011
}

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

print(myfamily)
```

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

[ ]:
```
Access Items in Nested Dictionaries
To access items from a nested dictionary, you use the name of the dictionaries,␣
 ↪starting with the outer dictionary:
```

[49]:
```
myfamily = {
  "child1" : {
    "name" : "Emil",
    "year" : 2004
  },
  "child2" : {
    "name" : "Tobias",
    "year" : 2007
  },
  "child3" : {
    "name" : "Linus",
    "year" : 2011
  }
}

print(myfamily["child2"]["name"])
```

```
Tobias
```

[ ]:
```
Dictionary Methods
Python has a set of built-in methods that you can use on dictionaries.

Method                                                  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
```

5.0.7 Control Statement in Python

[ ]:
```
Python Conditions and If statements
Python supports the usual logical conditions from mathematics:

Equals: a == b
Not Equals: a != b
Less than: a < b
Less than or equal to: a <= b
Greater than: a > b
Greater than or equal to: a >= b
These conditions can be used in several ways, most commonly in "if statements"␣
 ↪and loops.

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

[50]:
```
a = 33
b = 200
if b > a:
  print("b is greater than a")
```

```
b is greater than a
```

[ ]:
```
Elif
The elif keyword is Python's way of saying "if the previous conditions were not␣
 ↪true, then try this condition".
```

[51]:
```
a = 33
b = 33
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
```

```
a and b are equal
```

[ ]:
```
Else
The else keyword catches anything which isn't caught by the preceding␣
 ↪conditions.
```

[52]:
```
a = 200
b = 33
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")
```

```
a is greater than b
```

[53]:
```
# You can also have an else without the elif:


a = 200
b = 33
if b > a:
  print("b is greater than a")
else:
  print("b is not greater than a")
```

```
b is not greater than a
```

[ ]:
```
Short Hand If
If you have only one statement to execute, you can put it on the same line as␣
 ↪the if statement.
```

[54]:
```
if a > b: print("a is greater than b")
```

```
a is greater than b
```

[ ]:
```
Short Hand 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:
```

[55]:
```
a = 2
b = 330
print("A") if a > b else print("B")
```

```
B
```

[ ]:
```
This technique is known as Ternary Operators, or Conditional Expressions.

You can also have multiple else statements on the same line:
```

[56]:
```
a = 330
b = 330
print("A") if a > b else print("=") if a == b else print("B")
```

```
=
```

[ ]:
```
And
The and keyword is a logical operator, and is used to combine conditional␣
 ↪statements:
```

[57]:
```
a = 200
b = 33
c = 500
if a > b and c > a:
  print("Both conditions are True")
```

```
Both conditions are True
```

[ ]:
```
Or
The or keyword is a logical operator, and is used to combine conditional␣
 ↪statements:
```

[58]:
```
a = 200
b = 33
c = 500
if a > b or a > c:
  print("At least one of the conditions is True")
```

```
At least one of the conditions is True
```

[ ]:
```
Not
The not keyword is a logical operator, and is used to reverse the result of the␣
 ↪conditional statement:
```

[59]:
```
a = 33
b = 200
if not a > b:
  print("a is NOT greater than b")
```

```
a is NOT greater than b
```

[ ]:
```
Nested If
You can have if statements inside if statements, this is called nested if␣
 ↪statements.
```

[60]:
```
x = 41

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

```
Above ten,
and also above 20!
```

[ ]:
```
The 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.
```

[61]:
```
a = 33
b = 200

if b > a:
  pass
```

[ ]:
```
Python Loops
Python has two primitive loop commands:

while loops
for loops
```

[ ]:
```
The while Loop
With the while loop we can execute a set of statements as long as a condition␣
  ↪is true.
```

[62]:
```
i = 1
while i < 6:
  print(i)
  i += 1
```

```
1
2
3
4
5
```

[ ]:
```
The break Statement
With the break statement we can stop the loop even if the while condition is␣
  ↪true:
```

[63]:
```
i = 1
while i < 6:
  print(i)
  if i == 3:
    break
  i += 1
```

```
1
2
3
```

[ ]:
```
The continue Statement
```

With the continue statement we can stop the current iteration, and continue␣
↪with the next:

[64]:
```
i = 0
while i < 6:
  i += 1
  if i == 3:
    continue
  print(i)
```

```
1
2
4
5
6
```

[ ]:
```
The else Statement
With the else statement we can run a block of code once when the condition no␣
↪longer is true:
```

[65]:
```
i = 1
while i < 6:
  print(i)
  i += 1
else:
  print("i is no longer less than 6")
```

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

[ ]:
```
Python For Loops
A for loop is used for iterating over a sequence (that is either a list, a␣
↪tuple, a dictionary, a set, or a string).
```

[66]:
```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  print(x)
```

```
apple
banana
cherry
```

[ ]:
```
Looping Through a String
Even strings are iterable objects, they contain a sequence of characters:
```

[67]:
```
for x in "banana":
  print(x)
```

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

[68]:
```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  print(x)
  if x == "banana":
    break
```

```
apple
banana
```

[69]:
```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  if x == "banana":
    break
  print(x)
```

```
apple
```

[70]:
```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  if x == "banana":
    continue
  print(x)
```

```
apple
cherry
```

[ ]:
```
The range() Function
To loop through a set of code a specified number of times, we can use the␣
 ↪range() function,
```

[71]:
```
for x in range(6):
  print(x)
```

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

[72]:
```
#Using the start parameter:

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

```
2
3
4
5
```

[73]:
```
#Increment the sequence with 3 (default is 1):

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

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

[ ]:
```
Else in For Loop
The else keyword in a for loop specifies a block of code to be executed when␣
 ↪the loop is finished:
```

[74]:
```
for x in range(6):
  print(x)
else:
  print("Finally finished!")
```

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

[75]:
```
#Break the loop when x is 3, and see what happens with the else block:

for x in range(6):
  if x == 3: break
  print(x)
else:
```

```
  print("Finally finished!")
```

```
0
1
2
```

[ ]:
```
Nested Loops
A nested loop is a loop inside a loop.

The "inner loop" will be executed one time for each iteration of the "outer␣
 ↪loop":
```

[76]:
```
adj = ["red", "big", "tasty"]
fruits = ["apple", "banana", "cherry"]

for x in adj:
  for y in fruits:
    print(x, y)
```

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

5.0.8 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.
```

[ ]:
```
Creating a Function
In Python a function is defined using the def keyword:
```

[77]:
```
def my_function():
  print("Hello from a function")
```

[79]:
```
my_function()  # calling a function
```

```
Hello from a function
```

[ ]:
```
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:
```

[80]:
```
def my_function(fname):
  print(fname + " Refsnes")

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

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

[ ]:
```
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:

A parameter is the variable listed inside the parentheses in the function␣
 ↪definition.

An argument is the value that is sent to the function when it is called.
```

[ ]:
```
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.
```

[81]:
```
def my_function(fname, lname):
  print(fname + " " + lname)

my_function("Emil", "Refsnes")
```

```
Emil Refsnes
```

[ ]:
```
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.
```

```
This way the function will receive a tuple of arguments, and can access the␣
 ↪items accordingly:
```

[82]:
```
def my_function(*kids):
  print("The youngest child is " + kids[2])

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

```
The youngest child is Linus
```

[ ]:
```
Keyword Arguments
You can also send arguments with the key = value syntax.

This way the order of the arguments does not matter.
```

[83]:
```
def my_function(child3, child2, child1):
  print("The youngest child is " + child3)

my_function(child1 = "Emil", child2 = "Tobias", child3 = "Linus")
```

```
The youngest child is Linus
```

[ ]:
```
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:
```

[84]:
```
def my_function(**kid):
  print("His last name is " + kid["lname"])

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

```
His last name is Refsnes
```

[ ]:
```
Default Parameter Value
The following example shows how to use a default parameter value.

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

[85]:
```
def my_function(country = "Norway"):
  print("I am from " + country)

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

```
my_function("Brazil")
```

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

[ ]:
```
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.

E.g. if you send a List as an argument, it will still be a List when it reaches␣
 ↪the function:
```

[86]:
```
def my_function(food):
  for x in food:
    print(x)

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

my_function(fruits)
```

```
apple
banana
cherry
```

[ ]:
```
Return Values
To let a function return a value, use the return statement:
```

[88]:
```
def my_function(x):
  return 5 * x

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

```
15
25
45
```

[ ]:
```
Recursion
Python also accepts function recursion, which means a defined function can call␣
 ↪itself.
```

[89]:
```
def tri_recursion(k):
  if(k > 0):
    result = k + tri_recursion(k - 1)
    print(result)
```

```
  else:
    result = 0
  return result

print("\n\nRecursion Example Results")
tri_recursion(6)
```

```
Recursion Example Results
1
3
6
10
15
21
```

[89]: 21

5.0.9 Python Lambda

[ ]:
```
A lambda function is a small anonymous function.

A lambda function can take any number of arguments, but can only have one␣
 ↪expression.

Syntax
lambda arguments : expression
The expression is executed and the result is returned:
```

[90]:
```
x = lambda a: a + 10
print(x(5))
```

```
15
```

[ ]:
```
Lambda functions can take any number of arguments:
```

[91]:
```
x = lambda a, b : a * b
print(x(5, 6))
```

```
30
```

[92]:
```
x = lambda a, b, c : a + b + c
print(x(5, 6, 2))
```

```
13
```

[ ]:
```
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
```

[93]:
```
def myfunc(n):
  return lambda a : a * n

mydoubler = myfunc(2)

print(mydoubler(11))
```

```
22
```

[94]:
```
def myfunc(n):
  return lambda a : a * n

mytripler = myfunc(3)

print(mytripler(11))
```

```
33
```

[95]:
```
def myfunc(n):
  return lambda a : a * n

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

print(mydoubler(11))
print(mytripler(11))
```

```
22
33
```

[ ]:
```
---By Md Aquib---
```