Python3 Documentation

Table of Contents

Variables

We use variables to store values that can be used to control commands in our code. We can also alter these values throughout the code.

```
# Make a variable to store text
name = "Zach"


# Create variables that are numbers
num_one = 3
num_two = 4
sum = num_one + num_two


# We can also assign multiple variables at once
num_one, num_two = 3, 4


# The value of a variable can be changed after it has been created
num_one = num_one + 1
```

Variables have an associated ‘type’ based on their characteristics. A string (str) is a text element, an integer (int) is a whole number element, a float (float) is a number with decimal places, and a Boolean (bool) is an element that returns either True or False. We use the type command to determine the type of an element.

```
# The output will be str, or string
print(type(name))


# The output will be int, or integer
print(type(sum))


# The output will be bool, or boolean
print(type(3==3))



# We can change the type of an element by using the shortened name of the type
# We do this when concatenating strings, such as:


age = 16
print("My age is " + str(age))
```

User Input

We can use input from the user to control our code.

```
# Ask the user for input and save it to a variable to be used in code
# This will only work if the input is being used as a string
name = input("What is your name? ")

# If input needs to be used as a number (i.e for a mathematical
# calculation), include the term 'int' or 'float'
num_one = int(input("Enter a number: "))
num_two = int(input("Enter a second number: "))
num_three = float(input("Enter a third number: "))

# This input can then be used to control different parts of the code
print("Hello, " + name)
sum = num_one + num_two
```

Printing

We can print elements to the screen by using the print command. If we want to print text, we need to surround the text with quotation marks " ". Unlike previous versions of Python, Python3 requires parentheses for printing.

```
print("Hello world")
print(2 + 2)
print(10)
a=1
b=32
print(a+b)
print(a*b)
print(a**b)
print(b/a)
print(2*a+b)
print((a+b)*2)
```

Comments

We use comments to leave notes about the code to the reader. Comments are not actually run by Python; they are just there to help us read the code. We can make multiline comments with """ and single line comments with #.

```
"""
A multi-line comment describes your code
to someone who is reading it.
"""

Example:

"""
This program will ask the user for two numbers.
Then it will add the numbers and print the final value.
"""
number_one = int(input("Enter a number: "))
number_two = int(input("Enter a second number: "))
print("Sum: " + str(number_one + number_two))

# Use single line comments to clarify parts of code.

Example:

# This program adds 1 and 2
added = 1 + 2
print(added)
```

Break and Continue
We can use the break command to end our code. The continue command will leave the control structure at that point and move to the commands found afterward.

```
# This code will end when the use enters a negative number or 42
number = int(input("Enter a number: "))
while number != 42:
    if number < 0:
        break
    else:
        print(number)
    number = int(input("Enter a number: "))



# This code will only print the numbers 0 to 3 and 6
for i in range(5):
    if i < 4:
        print(i)
    else:
        continue
print(6)
```

Random Numbers
To be able to use the randint or choice functions, you must use import random at the beginning of your code.

```
# Random integer between (and including) low and high
import random
random_num = random.randint(low, high)
random_element = random.choice(string)


# Example:
# Returns random number within and including 0 and 10.
random_num = random.randint(0,10)


# Random element in a string
random_element = random.choice('abcdefghij')
```

Exceptions

Exception handling allows us to prevent our programs from crashing in the event of a fault.

```
# Try/Except with input
try:
    my_number = int(input("Enter an integer: "))
    print("Your number: " + str(my_number))
except ValueError:
    print("That wasn't an integer!")

# Try/Except for Type Errors
try:
    my_number = '2' + 2
except TypeError:
    print("A type error has occurred!")

# Try/Except for Key Errors
dictionary = {'1':'k', '3':'A', '4':'R', '5':'E', '6':'L'}
try:
    dictionary['2']
except KeyError:
    print("Key error")

# Try/Except for Attribute Errors
try:
    dictionary.no_method()
except AttributeError:
    print("Attribute Error!")

# You can also have
try:
    my_number = int(input("Enter an integer: "))
    print("Your number: " + str(my_number))
except:
    print("There was an error.")
```

If Statements

Use an if statement to instruct the computer to do something only when a condition is true. If the condition is false, the command indented underneath will be skipped.

```
if BOOLEAN_EXPRESSION:
    print("This executes if BOOLEAN_EXPRESSION evaluates to True")


# Example:


# The text will only print if the user enters a negative number
number = int(input("Enter a number: "))
if number < 0:
    print(str(number) + " is negative!")
```

If/Elif/Else Statements

Use an if/else statement to force the computer to make a decision between multiple conditions. If the first condition is false, the computer will skip to the next condition until it finds one that is true. If no conditions are true, the commands inside the else block will be performed.

```
if condition_1:
    print("This executes if condition_1 evaluates to True")
elif condition_2:
    print("This executes if condition_2 evaluates to True")
else:
    print("This executes if no prior conditions evaluate to True")


# Example:


# This program will print that the color is secondary
color == "purple"
if color == "red" or color == "blue" or color == "yellow":
    print("Primary color.")
elif color == "green" or color == "orange" or color == "purple":
    print("Secondary color.")
else:
    print("Not a primary or secondary color.")
```

Mathematical Operators

Use mathematical operators to alter values.

```
+   Addition
-   Subtraction
*   Multiplication
/   Division
**  Power
%   Modulus (Remainder)
()  Parentheses (For order of operations)


# Examples
z = x + y
w = x * y


# Division
a = 5.0 / 2                    # Returns 2.5
b = 5.0 // 2                   # Returns 2.0
c = 5/2                        # Returns 2.5
d = 5 // 2                     # Returns 2

# Increment (add one)
x += 1

# Decrement (subtract one)
x -= 1

# Absolute value
absolute_value = abs(x)

abs_val = abs(-5)              # Returns 5

# Square root
import math
square_root = math.sqrt(x)

# Raising to a power
```

```
power = math.pow(x, y)          # Calculates x^y
# Alternate
power = x**y                    # Calculates x^y


# Rounding
rounded_num = round(2.675, 2)   # Returns 2.68
```

Comparison Operators

Use comparison operators to compare elements in order to make decisions in your code. Comparison operators return booleans (True/False).

```
x == y      # is x equal to y
x != y      # is x not equal to y
x > y       # is x greater than y
x >= y      # is x greater than or equal to y
x < y       # is x less than y
x <= y      # is x less than or equal to y


# Comparison operators in if statements
if x == y:
    print("x and y are equal")


if x > 5:
    print("x is greater than 5.")
```

Logical Operators

Use logical operators to check multiple conditions at once or one condition out of multiple.

```
# And Operator
and_expression = x and y


# Or Operator
or_expression = x or y


# You can combine many booleans!
boolean_expression = x and (y or z)
```

Loops
Loops help us repeat commands which makes our code much shorter. Make sure everything inside the loop is indented one level.

For Loops
Use for loops when you want to repeat something a fixed number of times.

```
# This for loop will print "hello" 5 times
for i in range(5):
    print("hello")


# This for loop will print out even numbers 1 through 10
for number in range(2, 11, 2):
    print(i)


# This code executes on each item in my_list
# This loop will print 1, then 5, then 10, then 15
my_list = [1, 5, 10, 15]
for item in my_list:
    print(item)
```

While Loops
Use while loops when you want to repeat something an unknown number of times or until a condition becomes false. If there is no point where the condition becomes false, you will create an infinite loop which should always be avoided!

```
# This program will run as long as the variable 'number' is greater than 0
# Countdown from from 10 to 0
number = 10
while number >= 0:
    print(number)
    number -= 1



# You can also use user input to control a while loop
# This code will continue running while the user answers ‘Yes’
continue = input("Continue code?: ")
while continue == "Yes":
    continue = input("Continue code?: ")
```

Functions
Writing a function is like teaching the computer a new word.
Naming Functions: You can name your functions whatever you want, but you can't have spaces in the function name. Instead of spaces, use underscores (_) like_this_for_example Make sure that all the code inside your function is indented one level!

Defining a Function
We define a function to teach the computer the instructions for a new word. We need to use the term def to tell the computer we're creating a function.

```
def name_of_your_function():
    # Code that will run when you make a call to
    # this function.

# Example:

# Teach the computer to add two numbers
num_one = 1
num_two = 2
def add_numbers():
    sum = num_one + num_two
```

Calling a Function
We call a function to tell the computer to actually carry out the new command.

```
# Call the add_numbers() function once
# The computer will return a value of 3
add_numbers()



# Call the add_numbers() function 3 times and print the output
# The output will be the number 3 printed on 3 separate lines
print(add_numbers())
print(add_numbers())
print(add_numbers())
```

Returning Values in Functions
We can use the command return to have a function give a value back to the code that called it. Without the return command, we could not use any altered values that were determined by the function.

```
# We add a return statement in order to use the value of the sum variable
num_one = 1
num_two = 2
def add_numbers():
    sum = num_one + num_two
    return sum
```

Using Parameters in Functions
We can use parameters to alter certain commands in our function. We have to include arguments for the parameters in our function call.

```
# In this program, parameters are used to give two numbers
def add_numbers(num_one, num_two):
    sum = num_one + num_two
    return sum

# We call the function with values inside the parentheses
# This program will print ‘7’
print(add_numbers(3, 4))
# If we have a list with the same number of parameters, we
# can use the items to assign arguments using an asterisk
my_list = [3, 4]
print(add_numbers(*my_list))
```

Strings
Strings are pieces of text. We can gain much information about strings and alter them in many ways using various methods.

Indexing a String
We use indexing to find or take certain portions of a string. Index values always start at 0 for the first character and increase by 1 as we move to the right. From the end of the string, the final value also has an index of -1 with the values decreasing by 1 as we move to the left.

```
# Prints a character at a specific index
my_string = "hello!"
print(my_string[0])       # print("h")
print(my_string[5])       # print("!")

# Prints all the characters after the specific index
my_string = "hello world!"
print(my_string[1:])      # print("ello world!")
print(my_string[6:])      # prints("world!")

# Prints all the characters before the specific index
my_string = "hello world!"
print(my_string[:6])     # print("hello")
print(my_string[:1])     # print("h")

# Prints all the characters between the specific indices
my_string = "hello world!"
print(my_string[1:6])      # print("ello")
print(my_string[4:7])      # print("o w")

# Iterates through every character in the string
# Will print one letter of the string on each line in order
my_string = "Turtle"
for c in my_string:
    print(c)

# Completes commands if the string is found inside the given string
my_string = "hello world!"
if "world" in my_string:
```

```
   print("world")


# Concatenation
my_string = "Tracy the"
print(my_string + " turtle")    # print("Tracy the turtle")

# Splits the string into a list of letters
my_string = "Tracy"
my_list = list(my_string)       # my_list = ['T', 'r', 'a', 'c', 'y']

# Using enumerate will print the index number followed by a colon and the
# word at that index for each word in the list
my_string = "Tracy is a turtle"
for index, word in enumerate(my_string.split()):
    print(str(index) + ": " + word)
```

String Methods

There are many methods that can be used to alter strings.

```
# upper: To make a string all uppercase
my_string = "Hello"
my_string = my_string.upper()     # returns "HELLO"

# lower: To make a string all lowercase
my_string = "Hello"
my_string = my_string.lower()     # returns "hello"

# isupper: Returns True if a string is all uppercase letters and False otherwise
my_string = "HELLO"
print(my_string.isupper())        # returns True

# islower: Returns True if a string is all lowercase letters and False otherwise
my_string = "Hello"
print(my_string.islower())         # returns False

# swapcase: Returns a string where each letter is the opposite case from original
my_string = "PyThOn"
my_string = my_string.swapcase()  # returns "pYtHoN"
```

```
# strip: Returns a copy of the string without any whitespace at beginning or end
my_string = "       hi there       "
my_string = my_string.strip()     # returns "hi there"

# find: Returns the lowest index in the string where substring is found
# Returns -1 if substring is not found
my_string = "eggplant"
index = my_string.find("plant")   # returns 3
index = my_string.find("Tracy")   # returns -1

# split: Splits the string into a list of words at whitespace
my_string = "Tracy is a turtle"
my_list = my_string.split()       # Returns ['Tracy', 'is', 'a', 'turtle']
```

Tuples
Tuples are immutable sequences of items.

Creating a Tuple
We create a tuple by listing items inside parentheses. We can include elements of any type.

```
# Make a new tuple named "my_tuple"
my_tuple = (1, 2, 3, 4, 5)

# Tuple with elements of different types
my_tuple = (0, 1, "Tracy", (1, 2))

# Tuple with single element
my_tuple = (3,)

# Tuple of tuples
my_tuple((0, 1), (2, 3))
```

Altering a Tuple
Due to the immutable nature of tuples, we cannot alter individual elements in the tuple but can perform various other tasks with them.

```
# Get the length of the tuple
print(len(my_tuple))


# Accessing elements within nested tuples
print(my_tuple[0][0])
print(my_tuple[1][0])


# Concatenating tuples
x = (1, 2)
y = (5, 6)
my_tuple = x + (3,) + y
```

Lists
Lists are mutable sequences of items.

Creating a List
We create a list by listing items inside square brackets. We can include elements of any type.

```
# Create an empty list
my_list = []


# Create a list with any number of items
my_list = [item1, item2, item3]
# Example:
number_list = [1, 2, 4]


# A list can have any type
my_list = [integer, string, boolean]
# Example:
a_list = ["hello", 4, True]
```

Altering a List

Due to the mutable nature of lists, we can alter individual elements in the list.

```
# Access an element in a list
a_list = [“hello”, 4, True]
first_element = a_list[0]    # Returns "hello"

# Set an element in a list
a_list = [“hello”, 4, True]
a_list[0] = 9                # Changes a_list to be [9, 4, True]

# Looping over a list
# Prints each item on a separate line (9, then 4, then True)
a_list = [9, 4, True]
for item in a_list:
    print(item)

# Length of a list
a_list = [9, 4, True]
a_list_length = len(a_list)  # Returns 3

# Creates a list based on first operation
# This will create a list with numbers 0 to 4
a_list = [x for x in range(5)]
# This will create a list with multiples of 2 from 0 to 8
list_of_multiples = [2*x for x in range(5)]
```

List Methods

There are many methods that can be used to alter lists.

```
# append: Add to a list
a_list = ["hello", 4, True]
a_list.append("Puppy")       # Now a_list = ["hello", 4, True, "Puppy"]

# pop: Remove and return last element from the list
a_list = ["hello", 4, True]
last_item = a_list.pop()     # Removes True, now a_list = ["hello", 4]
# Remove and return an item from a list at index i
a_list = ["hello", 4, True]
a_list.pop(0)                # Removes "hello", now a_list = [4, True]

# index: Returns the index value of the first item in the list that matches element
# There is an error if there is no such item
a_list = ["hello", 4, True]
a_list.index(4)               # Returns 1 because 4 is found at index[1]
a_list.index("hi")            # Error because no item "hi"

# sort: Returns a sorted list
my_list = [9, 7, 1, 2, 3]
my_list.sort()               # Returns [1, 2, 3, 7, 9]

# reverse: Returns a reversed list
my_list = [1, 2, 3, 4]
my_list.reverse()            # Returns [4, 3, 2, 1]

# count: Returns the number of instances of a particular item that were found
my_list = [1, 4, 2, -4, 10, 0, 4, 2, 1, 4]
print(my_list.count(4))       # Returns 3
print(my_list.count(123))     # Returns 0 because 123 does not exist in list

# extend: Allows us to add a list to a list
my_list = [1, 2, 3]
my_list.extend([4, 5, 6])    # Returns [1, 2, 3, 4, 5, 6]
```

```
# remove: Allows us to remove a particular item from a list
# Only removes the first instance of the item
my_list = ["apple", "banana", "orange", "grapefruit"]
my_list.remove("orange")     # Returns ["apple", "banana", "grapefruit"]



# join: Creates string out of list with specified string placed between each item
my_list = ["Tracy", "is", "a", "turtle"]
(" ").join(my_list)          # Returns the list as a string with spaces between words
```

2D Lists
2D Lists allow us to create lists of lists.

```
# Create an empty list
my_list = []

# Add to the list
my_list.append([1, 2, 3])
my_list.append([4, 5, 6])

# Access elements within the nested lists
print(my_list[0])       # Returns [1, 2, 3]
print(my_list[0][1])    # Returns 2

# Take a slice of the outer list
print(my_list[0:2])     # Returns [[1, 2, 3], [4, 5, 6]]

# Take a slice of the inner list
print(my_list[0][0:2])  # Returns [1, 2]
```

Dictionaries
Dictionaries have a collection of key-value pairs.

```
a_dictionary = {key1:value1, key2:value2}
# Example:
my_farm = {pigs:2, cows:4}  # This dictionary keeps a farm's animal count

# Creates an empty dictionary
a_dictionary = {}

# Inserts a key-value pair
a_dictionary[key] = value
my_farm["horses"] = 1      # The farm now has one horse

# Gets a value for a key
my_dict[key] # Will return the key
my_farm["pigs"]            # Will return 2, the value of "pigs"

# Using the 'in' keyword
my_dict = {"a": 1, "b": 2}
print("a" in my_dict)       # Returns True
print("z" in my_dict)       # Returns False
print(2 in my_dict)        # Returns False, because 2 is not a key

# Iterating through a dictionary
for key in my_dict:
    print("key: " + str(key))
    print("value: " + str(my_dict[key]))
```

Classes

Classes hold multiple functions.

```
# Declare a class
class MyClass:
    # The __init__ method is called whenever we instantiate our class
    def __init__(self):
        print("Class initiated")
        self.my_num = 0

# Instantiate your class
my_class = MyClass()

# Access instance variables in your class
print(my_class.my_num)

my_class.my_num = 10

# Adding arguments to your class
class Point:
    def __init__(self, x = 0, y = 0):
        self.x = x
        self.y = y

# Instantiate the class
p = Point(3, 4)
```

Sets

A set contains an unordered collection of unique and immutable objects.

```
# Make a new set named "new_set"
new_set = set([])
girl_scout_badges = set([])

# Add to a set
new_set.add(item)
girl_scout_badges.add("Squirrel Whisperer")

# Does a set contain a value
item in my_set # Returns a boolean
"Squirrel Whisperer" in girl_scout_badges  # Returns True

# Number of elements in the set
len(my_set)
len(girl_scout_badges)  # Returns 1 since there is only one item in the set
```