Day 02

Strings in Python

String literals can be defined with either single or double quotes (your choice), and can be defined on multiple lines with a backslash like so:

```
s = 'broadridge'
k = "framework"
x = "my favorite " \
    "string"
```

A Python string is an iterable series of characters. You can loop through a string just like a list:

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

Most importantly, strings in Python are immutable. This means you cannot change strings like so:

```
my_str = "can't touch this"
my_str[6] = " " # TypeError
```

Also, when you build strings with += in a loop, you're creating a new string every iteration:

```
new_str = "hello " for c in "world":
new_str += c # new string created every single time! print(new_str) # hello world
```

Unicode vs ASCII

In Python 2 strings, are stored internally as 8 bit ASCII. But in Python 3, all strings are represented in Unicode.

Uh, what?

Before we talk about methods on strings in Python, let's learn a little bit about the history of character encodings. If you would like a longer description, feel free to read this excellent article.

When we as humans see text on a computer screen, we are viewing something quite different than what a computer processes. Remember that computers deal with bits and bytes, so we need a way to encode (or map) characters to something a computer can work with. In 1968, the American Standard Code for Information Interchange (or ASCII) was standardized as a character encoding. ASCII defined codes for characters ranging from 0 to 127.

Why this range? Remember that computers work in base 2 or binary, so each bit represents a power of two. This means that 7 bits can get us 2^7 = 128 different binary numbers; since each bit can equal0or1, with7bits we can represent all numbers from 0000000 up to 1111111. With ASCII, we can then map each of these numbers to a distinct character. Since there are only 26 letters in the alphabet (52 if you care about the distinction between upper and lower case), plus a handful of digits and punctuation characters, ASCII should more than cover our needs, right?

ASCII was a great start, but issues arose when non English characters like é or ö could not be processed and would just be converted to e and o. In the 1980s, computers were 8- bit machines which meant that bytes now held 8 bits. The highest binary number we could obtain on these machines was 11111111 or 2^0 + 2^1 + 2^2 + 2^3 + 2^4 + 2^5 + 2^6 + 2^7, or 255. Different machines now used the values of 128 to 255 for accented characters, but there was not a standard that emerged until the International Standards Organization (or ISO) emerged.

Even with an additional 128 characters, we started running into lots of issues once the web grew. Languages with completely different character sets like Russian, Chinese, Arabic, and many more had to be encoded in completely different character sets, causing a nightmare when trying to deliver a single text file with multiple character sets.

In the 1980s, a new encoding called Unicode was introduced. Unicode's mission was to encode all possible characters and still be backward compatible with ASCII. The most popular character encoding that is dominant on the web now is UTF-8, which uses 8- bit code units, but with a variable length to ensure that all sorts of characters can be encoded.
In Python3, strings are Unicode by default.

String Methods

Python contains quite a few helpful string methods; here are a few. Try running these in a REPL to see what they do!

Let's start with a simple variable:

```
1 string = "this Is nIce"
```

Upper

To convert every character to upper-case we can use the upper function.

```
string.upper() # 'THIS IS NICE'
```

Lower

To convert every character to lower-case we can use the lower function.

```
string.lower() # 'THIS IS NICE'
```

Capitalize

To convert the first character in a string to upper-case and everything else to lower-case we can use the capitalize function.

```
string.capitalize() # 'This is nice'
```

Title

To convert every first character in a string to upper-case and everything else to lower- case we can use the title function.

```
string.title() # 'This Is Nice'
```

Find

To find a subset of characters in a string we can use the find method. This will return the index at which the first match occurs. If the character/characters is/are not found, find will return -1

```
instructor = 'rajath'
instructor.find('r') # 0
instructor.find('R') # -1 it IS case sensitive!
string.find("j") # 2, since the character "j" is at index 2
string.find('kum') # -1
```

isalpha

To see if all characters are alphabetic we can use the isalpha function.

```
string.isalpha() # False
string[0].isalpha() # True
```

isspace

To see if a character or all characters are empty spaces, we can use the isspace function

```
string.isspace() # False
string[0].isspace() # False
string[4].isspace() # True
```

islower

To see if a character or all characters are lower-cased, we can use the islower function (there is also a function, which does the inverse called isupper)

```
string.islower() # False
string[0].islower() # True
string[5].islower() # False
string.lower().islower() # True
```

istitle

To see if a string is a "title" (first character of each word is capitalized), we can use the istitle function.

```
string.istitle() # False
string.title().istitle() # True
"not Awesome Sauce".istitle() # False
"Awesome Sauce".istitle() # True
```

Endswith

To see if a string ends with a certain set of characters we can use the endswith function.

```
"string".endswith('g') # True
"awesome".endswith('foo') # False
```

partition

To partition a string based on a certain character, we can use the partition function.

```
string.partition('i') # what's the type of what you get back?
"awesome".partition('e') # ('aw', 'e', 'some')
```

String Interpolation with Formatted Strings

One of the most common string methods you'll use is theformatmethod. This is a powerful method that can do all kinds of string manipulation, but it's most commonly just used to pass varaibles into strings. In general this is preferred over string concatenation, which can quickly get cumbersome if you're mixing a lot of variables with strings. For example:

```
first_name = "Rajath"
last_name = "Kumar"
city = "Bangalore"
mood = "great"
greeting = "Hi, my name is " + first_name + " " + last_name + ", I live in " + city + " and I feel " + mood + "."
greeting # 'Hi, my name is Rajath Kumar, I live in Bangalore and I feel great.'
```

Here, the greeting variable looks fine, but all that string concatenation isn't easy on the eyes. It's very easy to forget about a + sign, or to forget to separate words with extra whitespace at the beginning and end of our strings.

This is one reason why format is nice. Here's a refactor:

```
greeting = "Hi, my name is {} {}, I live in {} and I feel {}.".format(first_n ame, last_name, city, mood)
```

When we call format on a string, we can pass variables into the string! The variables will be passed in order, wherever format finds a set of curly braces.

Starting in Python 3.6, however, we have f-strings, which are a cleaner way of doing string interpolation. Simply put f in front of the string, and then brackets with **actual variable names**.

```
greeting = f"Hi, my name is {first_name} {last_name}, I live in {city}, and I feel {mood}."
```

Functions, Inbuilt Functions & Modules

What is a function?

A function is a repeatable process or procedure. A real world analogy of a function is the brew button on a coffee machine. The coffee machine has inputs (hot water, coffee grounds), and outputs (hot coffee). When you press the button to brew a pot of coffee, you are starting a process that should return an expected output to you. The same thing is true in programming. A function takes a set of variables as inputs and returns a value as an output.

We have already seen many functions in action. For example, in the list chapter, we learned about append and many others. These are built-in functions that operate on a list. But in addition to built in-functions, we can also write our own functions! In Python, a function has the following format:

```
def function_name():
    # code gets indented here
```

> **Note**
>
> Notice that we MUST indent on the following line. If you do not indent your code, you'll get an IndentationError! To invoke a function, use the ():

```
def first_function():
    print("Hello World!")
first_function() # Hello World!
```

Next, let's try to write a function called add_five_plus_five which outputs the sum of 5 + 5. Here's what that might look like:

```
def add_five_plus_five():
    5+5
```

Now let's run this function and our output is....nothing! Why is that? We are missing a very important keyword:

In order to output values from a function, we need to use the return keyword. Let's see how we can fix our function now.

```
def add_five_plus_five():
    return 5+5
```

Now let's run this function add_five_plus_five() and our output is....10! If we would like, we can also save this information to a variable and use it at a later point in time like this:

```
ten = add_five_plus_five()
print(ten + 10) # 20
```

If we don't have a return statement in our function, it will always return None to us. This is true regardless of what else happens in the function. Take a look at this example:

```
def print_five_plus_five():
    print(5 + 5)
def add_five_plus_five():
    return 5 + 5
ten = add_five_plus_five()
maybe_ten = print_five_plus_five() # this line should print 10 to the console ten # 10
maybe_ten # None
```

In the real world, we'd never really write functions like these because they are very rigid; all they do is add 5 and 5. Ideally, we'd like to be able to provide some input to our functions, but in order to do that we need to introduce a concept called parameters or arguments.

Function Parameters

Function with Arguments

Here is an example of a function that takes two arguments:

```
def pet_names(cat_name, dog_name):
    return f"I have a cat named {cat_name} and a dog named {dog_name}."
```

In this case, our function takes two arguments (a cat_name and a dog_name), and returns a message about the pets.

Keyword Arguments

One nice thing about Python is that if you know the names of the arguments that you will pass to a function, you can pass them into the function in any order. All you need to do is provide the name (or keyword) for the argument, then the value you want to pass in. Check it out:

```
def pet_names(cat_name, dog_name):
    return f"I have a cat named {cat_name} and a dog named {dog_name}."

# no keyword arguments - order matters!
pet_names("Mittens", "Fido") # "I have a cat named Mittens and a dog named Fi do."
pet_names("Fido", "Mittens") # "I have a cat named Fido and a dog named Mitte ns." -- uh oh, the names are the opposite of what we want, because we passed them to the function in the wrong order.

# keyword arguments
pet_names(cat_name="Mittens", dog_name="Fido")
# "I have a cat named Mittens and a dog named Fido."

# keyword arguments - order doesn't matter!
pet_names(dog_name="Fido", cat_name="Mittens")
# "I have a cat named Mittens and a dog named Fido."
```

When you call a function by passing in akeyword=valuepair, you're said to be using keyword arguments. This can be especially useful if you have a function that accepts many parameters.

Default argument values

Sometimes you may want to set default values for parameters you pass into your function. In Python, the syntax looks the same as when you use keyword arguments, with one crucial difference: you use keyword arguments when you call a function, but you use default argument values when you define a function. Here's an example:

```
def add(a=5,b=15):
    return a + b
```

In this case, if we don't pass in any values when we call add, the first parameter will be 5 and the second parameter will be 15. But we can overwrite these defaults by simply passing numbers into the function when we call it:

```
add(15,1) # 16
add(4) # 19 - a is set to 4, b is set to 15
add() # 20
add(b=30) # 35 - a is set to 5 by default and b is 30 using keyword arguments
```

Variable number of function arguments

Sometimes we might want to write a function that can be called with an unknown number of arguments. There are two ways we can do this. The first is by using *, in the function definition which allows us to pass in an unknown number of arguments:

```
def foo(*args):
    print(args)

foo(1,2,3) # (1,2,3)
foo(1,2) # (1,2)
foo([1,2,3]) # ([1,2,3])
```

Inside of the function, the named parameter after the * corresponds to a tuple of the arguments passed in.

This can be helpful if we want to iterate through all of the arguments or apply some other function on a tuple:

```
def add(*nums):
    return sum(nums)

add(1,2,3,4) # 10
```

We can also use the * operator when invoking a function. In that case, the `*` will take an iterable like a list and split it up into separate parameters. Here is an example:

```
def add_three_nums(n1, n2, n3):
    return n1 + n2 + n3
add_three_nums(*[5,6,4]) # same as add_three_nums(5,6,4)
```

Unpacking an argument

The same way that we can include a * before a parameter, we can also do this for values passed to a function. If you're coming from JavaScript, this is going to look very similar to the spread operator. We unpack arguments when we need to convert a collection (tuple / list) to comma separated values. The idea here would be, we want to invoke a function, but all we have is a collection - let's see an example.

```
def add_and_multiply_numbers(a,b,c): return a + b * c
numbers = [1,2,3]
more_numbers = (4,5,6)
add_and_multiply_numbers(numbers) # TypeError
add_and_multiply_numbers(*numbers) # 7

add_and_multiply_numbers(more_numbers) # TypeError
add_and_multiply_numbers(*numbers) # 34
```

When there is a `*` as a parameter to a function, that parameter will be a tuple of values when the function is invoked. When you find a `*` that is not a parameter to a function, we are unpacking a value.

Variable number of keyword arguments

What if you want to pass in an unknown number of keyword arguments? In this case, we can use **, which allows us to access all of the keyword arguments inside of a function as a dictionary when we do not know how many keyword arguments will be passed.

```
def print_kwargs(a,b,**kwargs):
    print(a,b,kwargs)
print_kwargs(1,2,awesome='sauce', test='yup') # 1 2 {'test': 'yup', 'awesome ':
'sauce'}
```

Unpacking a dictionary into keyword arguments

The same way that we can include a `**` before a parameter, we can also do this for values passed to a function. Previously, we saw that `*` will unpack a collection (list / tuple), so what about `**`? That's for dictionary unpacking! What the `**` operator will do is turn a dictionary into keyword arguments so that if we have one single dictionary, we can pass it to a function and unpack it into keyword arguments! Since dictionaries do not guarantee any kind of order, it's useful that they become keyword arguments since a function that uses keyword arguments can accept those keyword arguments in any order!

```
def add_and_multiply_numbers(a,b,c):
    return a + b * c
data = dict(a=1,b=2,c=3)
add_and_multiply_numbers(data) # TypeError
add_and_multiply_numbers(**data) # 7
```

When there is a `*` as a parameter to a function, that parameter will be a tuple of values when the function is invoked. When you find a `*` that is not a parameter to a function, we are unpacking a value.

Function Scope

Scope

In Python we have function scope, which prohibits us from accessing variables created inside of a function from outside of that function:

```
def func(): x=5
    return x
func() # 5
x # NameError
```

The global keyword

The global scope includes all variables defined outside of functions. But if we try to use a global variable in a method, we will see UnboundLocalError: local variable VARIABLE_NAME referenced before assignment. This happens because a method in Python either has local variables or global variables. If variable is defined anywhere in a method and that variable has the same name as a global variable, then the new local variable will be used in the function instead of the global. But if you actually want to assign a global variable from within a function, you need to use the global keyword.
Using global variables in general is not best practice:

```
id = 0
def increment_id():
  id += 1
increment_id() # UnboundLocalError: local variable 'id' referenced before assignment

def increment_id():
  global id
  id += 1
increment_id() # The global id is now 1
```

In Python you need to explicitly state that a variable should be global, using the `global` keyword.

Listing Locals and Globals

In Python we can display all of the local variables and global variables using the `locals` and `globals` functions

```
def print_locals():
    x=2
    name = "rajath"
    print(locals())

name = "kumar"
print(globals())
print(locals())
```

Python Nested Functions (sort of like closures)

In Python we do have support for closures, a feature where an inner function has access to variables in an outer function's scope, even after the outer function has finished executing.

```
def outer(a):
    def inner(b):
        return a + b
    return inner

outer(3)(4) # 7
x = outer(2)
x(10) # 12
```

However, closures in Python are "weak" and have some limitations. For example, if you want to change the value of a variable from an outer scope, you'll run in to problems:

```
def counter():
    x = 0

    def increment():
        x += 1
        print(x)

    return increment

counter()() # UnboundLocalError: local variable 'x' referenced before assignment
```

Again, this is because the x inside of increment is a new variable, bound to the scope of increment. It's not a reference to x coming from the scope of counter.

We can get around the problem with the example above by setting attributes on the inner function, rather than trying to change variables from an outer scope:

```
# We can get around this by doing
def outer_count():

def inner_count():
    inner_count.x += 1
    print(inner_count.x)

    inner_count.x = 0

    return inner_count
```

Documenting our functions

Something that Python offers us is the ability to add what is called a docstring. Let's see what that looks like

```
def say_hello():
    # we are using three quotes so that this can be a multi-line string if necessary
    """This function returns the string hello when called"""
    return "hello"
```

We can call this function using

```
say_hello() # "hello"
say_hello.__doc__ # "This function returns the string hello when called"
help(say_hello) # gives us even more detail with the docstring!
```

Docstrings are essential when writing methods and can be thought of like an enhanced comment. Docstrings are also very useful when writing tests, as you can see what the docstring is when running the test. You are highly encouraged to write docstrings for your functions, and inside classes as well.

Default argument types for Python

Unlike languages like Java and C++, Python is a dynamically typed language. This means that we do not need to explicitly define the data type of a variable when initializing it. This gives us a bit more flexibility around our code, but sometimes we want to clearly indicate that a certain data type is what should be passed as a parameter, or that a function returns a specific value. We can do that in Python! Let's see what that looks like:

```
def add(a: int, b: int) -> int:
  """This function returns the sum of two numbers"""
  return a + b
```

We are specifying that both a and b are ints and the return value from the function is an int as well. We can also combine this with default parameter values!

```
def add(a: int = 5 ,b: int = 5) -> int:
  """This function returns the sum of two numbers with default values of 5
  for a and 5 for b"""
  return a + b
```

Popular Inbuilt functions of Python

len()

Returns the number of items in an object.

```
myList = [1, 2, 3, 4]
print(len(myList))  # Output: 4
```

print()

Prints the given object(s) to the standard output device.

```
print("Hello, World!")  # Output: Hello, World!
```

type()

Returns the type of an object.

```
print(type(123))  # Output: <class 'int'>
```

int(), float(), str()

Converts objects to an integer, floating point number, or string, respectively.

```
print(int("10"))    # Output: 10
print(float("10"))  # Output: 10.0
print(str(10))      # Output: "10"
```

input()

Allows user input.

```
name = input("Enter your name: ")
print(name)
```

max()

Returns the largest item.

```
print(max(1, 3, 2))  # Output: 3
```

min()

Returns the smallest item.

```
print(min(1, 3, 2))  # Output: 1
```

sum()

Sums up the items.

```
numbers = [1, 2, 3]
print(sum(numbers))  # Output: 6
```

range()

Generates a sequence of numbers.

```
for i in range(5):
    print(i)  # Output: 0 1 2 3 4
```

sorted()

Returns a sorted list from the items in an iterable.

```
print(sorted([3, 1, 2]))  # Output: [1, 2, 3]
```

map()

Applies a function to all the items in an input list.

```
items = [1, 2, 3]
squared = list(map(lambda x: x**2, items))
print(squared)  # Output: [1, 4, 9]
```

filter()

Constructs an iterator from elements of an iterable for which a function returns true.

```
numbers = [1, 2, 3, 4, 5, 6]
even = list(filter(lambda x: x % 2 == 0, numbers))
print(even)  # Output: [2, 4, 6]
```

abs()

Returns the absolute value of a number.

```
print(abs(-5))  # Output: 5
```

round()

Rounds a number to a specified number of digits.

```
print(round(3.14159, 2))  # Output: 3.14
```

eval()

Evaluates a given expression string and executes it as Python code.

```
print(eval('3 + 4'))  # Output: 7
```

globals() and *locals()

Return the current global and local dictionary, respectively.

```
x = 5
print(globals())  # Output includes 'x': 5
def func():
    y = 10
    print(locals())  # Output: {'y': 10}
func()
```

help()

Invokes the built-in help system.

```
help(print)
```

id()

Returns the identity of an object.

```
x = 'hello'
print(id(x))  # Output: memory address of x
```

These examples cover a range of functionalities provided by Python's built-in functions, from basic data manipulation to more complex operations.

Popular Inbuilt Modules of Python

Let's explore some popular built-in modules in Python along with examples of how to use them:

math

Provides access to mathematical functions.

```
import math
print(math.sqrt(16))  # Output: 4.0
print(math.pi)        # Output: 3.141592653589793
```

datetime

For manipulating dates and times.

```
import datetime
now = datetime.datetime.now()
print(now)  # Output: Current date and time
```

os

Offers a way of using operating system dependent functionality.

```
import os
print(os.getcwd())  # Output: Current working directory
os.mkdir('new_directory')
```

sys

Provides access to some variables used or maintained by the Python interpreter.

```
import sys
print(sys.version)  # Output: Python version
```

json

For encoding and decoding JSON data.

```
import json
jsonData = '{"name": "John", "age": 30}'
pythonObj = json.loads(jsonData)
print(pythonObj)  # Output: {'name': 'John', 'age': 30}
```

random

Generates pseudo-random numbers.

```
import random
print(random.randint(1, 10))  # Output: Random number between 1 and 10
```

re

Provides regular expression matching operations.

```
import re
pattern = '^a...s$'
test_string = 'abyss'
result = re.match(pattern, test_string)
if result:
  print("Search successful.")  # Output if pattern is found in the test_string
else:
  print("Search unsuccessful.")
```

subprocess

Allows you to spawn new processes.

```
import subprocess
subprocess.run(['ls', '-l'])  # Run "ls -l" command in Unix/Linux
```

urllib

For opening and reading URLs.

```
import urllib.request
with urllib.request.urlopen('http://python.org/') as response:
   html = response.read()
```

collections

Implements specialized container datatypes.

```
from collections import Counter
c = Counter('hello world')
print(c)  # Output: Counter({'l': 3, 'o': 2, 'h': 1, 'e': 1, ' ': 1, 'w': 1, 'r': 1, 'd': 1})
```

itertools

Implements a number of iterator building blocks.

```
import itertools
for p in itertools.permutations('ABCD', 2):
    print(p)
# Output: All 2-length permutations of 'ABCD'
```

threading

For threading (running multiple threads).

```
import threading

def printNumbers():
    for i in range(1, 6):
        print(i)

t = threading.Thread(target=printNumbers)
t.start()
t.join()
```

multiprocessing

Offers both local and remote concurrency.

```
from multiprocessing import Process

def func():
    print('hello from another process')

p = Process(target=func)
p.start()
p.join()
```

socket

For network connections.

```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(("www.python.org", 80))
```

sqlite3

For working with SQLite databases.

```
import sqlite3
conn = sqlite3.connect('example.db')
c = conn.cursor()
c.execute('''CREATE TABLE stocks (date text, trans text, symbol text, qty real, price real)''')
conn.commit()
conn.close()
```

logging

Provides a flexible framework for emitting log messages.

```
import logging
logging.basicConfig(level=logging.DEBUG)
logging.debug('This message should appear on the console')
```

argparse

For writing user-friendly command-line interfaces.

```
import argparse
parser = argparse.ArgumentParser(description='Process some integers.')
parser.add_argument('integers', metavar='N', type=int, nargs='+',
                    help='an integer for the accumulator')
args = parser.parse_args()
print(args.integers)
```

time

For time-related functions.

```
import time
print(time.time())  # Output: current time in seconds since the Epoch
```

Lambda Functions

The closest we have to "anonymous" functions in Python is lambdas. Lambdas are useful if you want to write a function which can be described in a single line of code. Here are some examples:

```
add = lambda x,y: x + y
double = lambda val: 2 * val
yell = lambda str: str.upper() + "!!!"

add(1,2) # 3
double(5) # 10
yell("hello") # 'HELLO!!!'

add.__name__ # '<lambda>'
```

Lambda functions start with the keyword lambda. Next comes a comma separated list of arguments, then a colon, then the expression you want the lambda to return. For simple one-line functions, lambdas can be a convenient shorthand for the traditional function definition. But these functions are anonymous; as you can see, they all share the same name.

Conditional and Iterable Statements

Conditional Statements

Conditional statements are used to execute certain pieces of code based on some condition. The most common conditional statements in Python are if, elif (else if), and else.

if Statement

```
x = 10
if x > 5:
    print("x is greater than 5")
# Output: x is greater than 5
```

if-else Statement

```
x = 3
if x > 5:
    print("x is greater than 5")
else:
    print("x is not greater than 5")
# Output: x is not greater than 5
```

if-elif-else Statement

```
x = 10
if x < 5:
    print("x is less than 5")
elif x == 10:
    print("x is 10")
else:
    print("x is greater than 5 but not 10")
# Output: x is 10
```

Iterable Statements

Iterable statements allow you to execute a block of code multiple times. The most common iterable statements in Python are for and while loops.

For Loop

```
for i in range(5):
    print(i)
# Output: 0 1 2 3 4
```

you can also iterate over elements in a list:

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

While Loop

```
i = 0
while i < 5:
    print(i)
    i += 1
# Output: 0 1 2 3 4
```

Nested Loop

You can also nest loops within each other, combining both for and while loops.

```
for n in [8, 9]:  # Outer loop for the numbers 8 and 9
    for i in range(1, 11):  # Inner loop for the numbers 1 to 10
        print(f"{n} x {i} = {n * i}")
    print("-" * 10)  # Separator between the tables

```

break Statement

The break statement is used to exit a loop prematurely, breaking out of the enclosing `for` or `while` loop.

```
for i in range(1, 10):
    if i == 5:
        break
    print(i)
# Output: 1 2 3 4
```

In this example, the loop terminates when i becomes 5, and thus numbers from 5 to 9 are not printed.

continue Statement

The `continue` statement is used to skip the rest of the code inside a loop for the current iteration only. The loop does not terminate but continues with the next iteration.

```
for i in range(1, 10):
    if i == 5:
        continue
    print(i)
# Output: 1 2 3 4 6 7 8 9
```

Here, when i is 5, the continue statement is executed, which causes the loop to skip the print statement for 5 and proceed with the next iteration.

break ans continue in while Statement

```
i = 0
while i < 10:
    i += 1
    if i == 5:
        continue
    if i == 8:
        break
    print(i)
# Output: 1 2 3 4 6 7
```

In this while loop example, the loop skips printing the number 5 due to the continue statement, and it breaks out of the loop entirely when i becomes 8.

break and continue are fundamental to controlling the flow of loops in Python, allowing for more complex and efficient looping constructs.

Deep Dive into Data Structures (Lists, Dictionaries, Tuples, Sets)

Lists in Python

Lists in Python are simply collections of elements. They can be as long as you want, and the individual elements can have the same type or not:

```
number_list = [1, 2, 3, 4, 5]
string_list = ["a", "b", "c", "d"]
kitchen_sink_list = [4, "yo", None, False, True, ["another", "list"]]
```

To access an element in a list, we use bracket notation and pass in the index we're interested in. Lists in Python use a zero-based index:

```
my_list = ["a", 1, True]
my_list[0] # "a"
my_list[2] # True
my_list[3] # IndexError
```

Note that if you try to access an element with an invalid index, Python will give you an error.

We can also reassign values in lists using =:

```
my_list = ["a", 1, True]
my_list[2] = False
my_list # ["a", 1, False]
```

In addition to getting and setting values in lists, there are a number of built-in methods you can use:

1. **append(x)**: Adds a single element x to the end of the list.
2. **extend(iterable)**: Extends the list by appending all the items from the iterable.
3. **insert(i, x)**: Inserts an item x at a given position i.
4. **remove(x)**: Removes the first item from the list whose value is equal to x.
5. **pop([i])**: Removes the item at the given position in the list, and returns it. If no index is specified, pop() removes and returns the last item in the list.
6. **clear()**: Removes all items from the list.
7. **index(x[, start[, end]])**: Returns the index of the first item whose value is equal to x.
8. **count(x)**: Returns the number of times x appears in the list.

9. **sort(*, key=None, reverse=False)**: Sorts the items of the list in place (the arguments can be used for sort customization).
10. **reverse()**: Reverses the elements of the list in place.
11. **copy()**: Returns a shallow copy of the list.

Slicing Lists in Python

Slices return portions of a list or string. While this seems like a pretty minor concept, there's actually quite a bit you can do with slices that you might not expect.

```
first_list = [1,2,3,4,5,6]
first_list[0:1] # [1]

# if a value for end isn't provided, you'll slice to the end of the list
first_list[1:] # [2, 3, 4, 5, 6]

# if a value for start isn't provided, you'll slice from the start of the lis t
first_list[:3] # [1,2,3]

# get the last element in the list
first_list[-1] # 6

# start from the second to last element in the list
first_list[-2:] # [5, 6]

# There is always more than one way of doing something...
first_list[4:] == first_list[-2:] # True

# step in the opposite direction
first_list[::-1] # [6, 5, 4, 3, 2, 1]

# step in the opposite direction by two elements
first_list[::-2] # [6, 4, 2]
```

List Iterations and Comprehension

Iterating over lists and strings

In Python we have a few ways of iterating over lists and strings. One of the most common types of loops is a for in loop; while loops are also common. Let's see what those look like.

for in

The most common way of iterating over a list is a for in loop. The syntax is for ELEMENT in LIST:. As with if statements, don't forget about the colon!

```
values = [1,2,3,4]
for val in values:
    print(val)

for char in "awesome":
    print(char)
```

Sometimes you may want to have access to the element's index in the list as well as the element itself. In this case, you can pass the list into the enumerate function. You'll need to name two variables in the for loop: the first will refer to the current index, the second will refer to the current element:

```
for idx, char in enumerate("awesome"):
    print(idx, char)
```

while

You can also do a while loop with Python, but this is a bit less common when iterating:

```
i=0
while i < 5:
    print(i)
    i +=1
```

If you ever want to move to the next step of the iteration, you can prematurely break out of the current iteration with the the continue keyword. Similarly, you can exit from a loop entirely using the break keyword.

```
for num in [1, 2, 3, 4, 5, 6, 7]:
    if num % 2 == 0:
      continue
    elif num > 5:
      break
    print(num)

```

range

In Python we can also create ranges, which represent a range of numbers, with the following syntax: . Note that the range is not inclusive. In other words, will include 1, 2, and 3, but not 4!

```
# We can do some pretty cool things with range
(a,b,c,d) = range(4)

for num in range(4,10):
    print(num)

# Note that the chr functions takes in a number # and returns the ascii character for the number

capital_letters = []
```

```
for num in range(65,91):
    capital_letters.append(chr(num))
    capital_letters

# Output:['A','B','C','D','E','F','G','H','I','J',
#   'K','L','M','N','O','P','Q','R',
#   'S','T','U','V','W','X','Y','Z']
```

Ranges take up less memory than lists, so if you find yourself needing a bunch of numbers that increment by the same amount each time, try to use a range instead of a list.

List Comprehension

List comprehensions are one of the most powerful tools in Python. They allow you to build lists in a more concise way, often in a single line. List comprehensions are a wonderful alternative to loops!

One way to use a list comprehension is to transform a set of values from a range or another list into some new set of values. This is sometimes referred to as a mapping operation. Here are a few examples:

```
# return a list of squares
[num**2 for num in range(10)] # [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

[chr(num) for num in range(65,91)]
# Output:['A','B','C','D','E','F','G','H','I','J',
#   'K','L','M','N','O','P','Q','R',
#   'S','T','U','V','W','X','Y', 'Z']
```

We can also put `if` statements inside of our list comprehensions to filter out certain transformed values!

```
# option 1 without list comprehension
vowels = []
for letter in 'awesome':
    if letter in ['a','e','i','o','u']:
        vowels.append(letter)

print(vowels) # ['a', 'e', 'o', 'e']

# option 2 with list comprehension
# In this example, the first letter is the value that we want in the new list
# and the if portion is the filter step
vowels = [letter for letter in 'awesome' if letter in ['a','e','i','o','u']]
print(vowels) # ['a', 'e', 'o', 'e']

# Count of 3 letter words in a string
len([word for word in "the quick brown fox jumps over the lazy dog".split(" ") if
len(word) == 3])

```

For longer list comprehensions, we can also split it into multiple lines for readability:

```
len([
  word
  for word in "the quick brown fox jumps over the lazy dog".split(" ")
  if len(word) == 3
])
```

Dictionary in Python

A dictionary in Python is an unordered collection of items. While other compound data types have only value as an element, a dictionary has a key-value pair.

Characteristics:

- **Mutable**: You can change their content without changing their identity.
- **Dynamic**: They can grow and shrink as needed.
- **Nested**: You can nest dictionaries inside dictionaries, lists, etc.
- **Key-Value Pairs**: Each item is a pair (key, value).

Basic Operations

Creating a Dictionary

```
my_dict = {'name': 'Alice', 'age': 25}
```

Accessing Elements

```
print(my_dict['name'])  # Output: Alice
```

Adding or Modifying Elements

```
my_dict['age'] = 26  # Modify
my_dict['address'] = 'Downtown'  # Add
```

Deleting Elements

```
del my_dict['address']  # Remove entry with key 'address'
my_dict.pop('age')  # Remove entry with key 'age' and return its value
my_dict.clear()  # Clear all entries in the dictionary
```

Dictionary Comprehension

Dictionary comprehension is a concise and readable way to create dictionaries. It is similar to list comprehension but for dictionaries.

Syntax

```
{key: value for vars in iterable}
```

For Examples,

Creating a simple dictionary comprehension

```
squares = {x: x*x for x in range(6)}
print(squares)  # Output: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

Using conditionals in dictionary comprehension

```
odd_squares = {x: x*x for x in range(11) if x % 2 != 0}
print(odd_squares)  # Output: {1: 1, 3: 9, 5: 25, 7: 49, 9: 81}
```

Using multiple iterables

```
combined = {k: v for k in ['a', 'b', 'c'] for v in [1, 2, 3]}
print(combined)  # Output: {'a': 3, 'b': 3, 'c': 3}
```

Tuples in Python

Tuples are similar to lists in Python, but they are immutable, meaning they cannot be changed after they are created. Tuples are often used for data that should not be modified.

Characteristics:

- Immutable: Once a tuple is created, you cannot change its values.
- Ordered: The items have a defined order, and that order will not change.
- Indexed: Tuples can be indexed and sliced like lists.

Basic Operations of Tuples

Creating a Tuple

```
my_tuple = (1, 2, 3)
```

Accessing Elements in Tuples

```
print(my_tuple[1])  # Output: 2
```

Slicing Elements

```
print(my_tuple[1:])  # Output: (2, 3)
```

Iterating through Elements

```
for item in my_tuple:
    print(item)
# Output: 1 2 3
```

Concatination and Repetition

```
t1 = (1, 2, 3)
t2 = ('a', 'b')
print(t1 + t2)  # Output: (1, 2, 3, 'a', 'b')
print(t2 * 2)   # Output: ('a', 'b', 'a', 'b')
```

Sets in Python

Sets are unordered collections of unique elements. They are mutable and are often used for operations involving membership tests, removing duplicates from a sequence, and computing mathematical operations such as intersection, union, difference, and symmetric difference.

Characteristics:

- Unordered: The elements in a set do not have a defined order.
- Mutable: You can add or remove items from a set.
- Unique Elements: Each element in a set is unique; duplicates are not allowed.

Basic Operations of Sets

Creating a Set

```
my_set = {1, 2, 3}
```

Adding Element in Set

```
my_set.add(4)
```

Removing Element from Set

```
my_set.remove(2)
# or
my_set.discard(3)
```

Set Operations (Union, Intersection, Difference, Symmetric Difference)

```python
a = {1, 2, 3}
b = {3, 4, 5}

print(a | b)  # Union: {1, 2, 3, 4, 5}
print(a & b)  # Intersection: {3}
print(a - b)  # Difference: {1, 2}
print(a ^ b)  # Symmetric Difference: {1, 2, 4, 5}
```

Set Comprehension

```python
squared = {x**2 for x in range(4)}
print(squared)  # Output: {0, 1, 4, 9}
```

Tuples and sets are fundamental data structures in Python, each with their specific use-cases and characteristics. Tuples are used for ordered and immutable collections of items, whereas sets are used for unordered collections of unique elements, particularly for set operations.