# Python

## Contents

## Python

---

### Links

---

- Composing Programs
- Kaggle
- Seqouia Notes
- Python Textbooks

# Data Types

- python is **loosely** typed

### Multiple Assignment

```
x, y = 10, 20
(x, y) = (10, 20) # equivalent (can leave off parantheses)
#creating a tuple, and then looping over the tuple

#works with other iterables:
x, y = [10, 20]
x, y = 'hi'

#unpacking with for a loop:
for i, line in enumerate(my_file):
  ...

#multiple assignment:
numbers = [1, 2, 3, 4, 5, 6]
first, *rest = numbers
*beginning, last = numbers

#deep unpacking:
color, (x, y, z) = ("red", (1, 2, 3))
```

### Strings

```
#some comment
print("Hello World")

my_message = 'Hello World'
print(my_message)

```

```python
multi_line_string = """This sentence
spans multiple
lines"""

print(len(my_message)) # 11

#slicing
print(my_message[0])   # 'H'
print(my_message[0:5]) # [beginning, end), 'Hello'
print(my_message[:5])  # 'Hello'
print(my_message[6:])  # 'World'

new_message = my_message.replace('World', 'Universe')

#formatted strings
greeting = 'Hello'
name = 'Bob'
message = '{}, {}. Welcome!'.format(greeting, name)
message = '{1}, {0}. Welcome!'.format(name, greeting)
message = '{my_name}, {my_greeting}. Welcome!'
          .format(my_name=name, my_greeting=greeting)

#f-strings
message = f'{greeting}, {name.upper()}. Welcome!'

print('Test'*5) # 'TestTestTestTestTest'
```

- no semicolons, operates on indentation
- convention (PEP8) to use snake_case for variables
  - upper SNAKE_CASE for constants
  - CamelCase for class names
- can use single or double quotes
- triple quotes span multiple lines
- string methods:
  - `str.lower()`
  - `str.upper()`
  - `str.count(char or string)`

  - `str.find(char or string)`, returns index
  - `str.replace(a, b)`, does not replace in-place
- can concatenate strings with `+` and `+=`
  - can also use a formatted string
  - can also use an f-string in Python 3.6+
    - allows embedded string
  - cannot simply concatenate with number, type error
    - must use conversion functions:
    - `str()`, `int()`, `float()`
- can use string replication operator *
  - `string * int`
- `dir(r)` function displays methods available with var
  - `help()` gives overview of methods
- docstrings have format """...."""
  - like a comment
  - documenting a function

### Numbers

```
#integers vs. floats
num = 3
num = 3.14

#some operands:
print(3 / 2)  # does not truncate in Python 3
print(3 // 2) # DOES truncate, floor division
print(3 ** 2) # exponent
#no unary increment/decrement!

num_1 = '100'
num_2 = '200'
print(num_1 + num_2) # concatenates

num_1 = int(num_1)   # casting
num_2 = int(num_2)
print(num_1 + num_2) # adds
```

- some built-in number functions:
  - **abs**()
  - **round**(), can specify how many digits after decimal to round to

## Lists, Tuples, and Sets

### Lists

```
empty_list = []
empty_list = list()

courses = ['History', 'Math', 'Physics']

print(courses)
print(len(courses)) # 3
print(courses[0])   # 'History'
print(courses[-1])  # negative index count from back, 'Physics'

#slicing
print(courses[0:2]) # [beginning, end) 'History', 'Math'
print(courses[:2])  # 'History', 'Math'
print(courses[2:])  # 'Math', 'Physics'

#adding items
courses.append('Art')
courses.insert(0, 'Art')

courses_2 = ['Art', 'Education']
courses.insert(0, courses_2) # creates a nested first element
courses.extend(courses_2)    # extends individual items

#removing items
courses.remove('Math')
popped = courses.pop()

#'in' operator
print('Math' in courses) # true
print('Art' in courses)  # false
```

```
for item in courses:
  print(item)
for index, course in enumerate(courses, start=1):
  # enumerate gives index and value, can take starting index
  print(index, course)

course_str = ', '.join(courses)   # join to string
new_list = course_str.split(', ') # split to list
```

- other list functions:
  - len, any, all
  - `sorted(list)` does not sort in place
  - `min(list)`
  - `max(list)`
  - `sum(list)`
- other list methods:
  - `list.reverse()`
  - `list.sort()`
  - `list.sort(reverse=True)`
  - `list.index(val)` gives ValueError if not in list
    - `val in list`, `in` operator
- equality:
  - equal if and only if elements are same and in same order

## Tuples

```
empty_tuple = ()
empty_tuple = tuple()

#mutable
list_1 = ['History', 'Math', 'Physics']
list_2 = list_1
list_1[0] = 'Art' # both lists are changed

#immutable
tuple_1 = ('History', 'Math', 'Physics')
tuple_2 = tuple_1
tuple_1[0] = 'Art' # TypeError, tuples are immutable
```

- immutable, unlike lists, which are mutable
- convention with tuples to use with heterogeneous collections
  - similar to structs

### Sets

```
empty_set = {} # dictionary, NOT a set
empty_set = set()

courses = {'History', 'Math', 'Physics', 'Math'} # order can change
courses_2 = {'History', 'Math', 'Physics', 'Design'}

print(courses.intersection(courses_2))
```

- order in a set can change from execution to execution
  - rejects duplicates
  - optimized for doing *membership-tests*
- functions with multiple sets:
  - `intersection()`
  - `difference()`
  - `union()`

### Dictionaries

```
student = {'name': 'John', 'age': 25, 'courses': ['Math', 'Physics']}

print(len(student))
print(student.keys())
print(student.values())
print(student.items())) # gives key-value pairs

print(student['name'])
print(student['phone']) # KeyError

print(student.get('name'))
print(student.get('phone'))              # returns None instead of error
print(student.get('phone', 'Not Found')) # returns default value
```

```

#changing or updating values:
student['phone'] = '555-5555'
student['name'] = 'Jane'

student.update({'name': 'Jane', 'age': 26})

#removing key-value pair:
del student['age']
age = student.pop('age')

#iterating through:
for key in student:
  print(key) # only gives the key

for key, values in student.items(): # also keys() and values()
  print(key, value) # create a tuple

keys = list(student.keys()) # making a list
```

- collection of key-value pairs
- keys can be any immutable data type
- equality:
    - equal if elements are the same

## Control Flow

### If / Else

- booleans: `True`, `False`
- false values:
    - `False`, `None`
    - zero of any numeric type
    - any empty sequence
        - ’ ’, (), []
    - any empty mapping

    * {}

- boolean operations: **and**, **or**, **not**
- python blocks are organized by indentation:

```python
if True:
  print('conditional true')

language = 'Python'

if language == 'Python': # equality
  print('Language is Python')
elif language == 'Java':
  print('Language is Java')
else:
  print('No match')

user = 'Admin'
logged_in = True

if user == 'Admin' and logged_in:
  print('Admin Page')
else:
  print('Bad Creds')

if not logged_in: # not reverses a boolean
  print('Please Log In')

a = [1, 2, 3]
b = [1, 2, 3]
c = a

print(a == b) # True, lists are equal
print(a is b) # False, not the same list
print(a is c) # True
print(id(a) == id(b)) # checking memory location
```

- can test for object identity with **`is`**
  - test for same object in memory
- can chain comparison operators in python:
  - **`if`** `a < b < c:`
  - implies `a < b` and `b < c`
- Python does NOT have switch-case statement

### In Keyword

- **`in`** checks for memberships

```
'in' in 'indigo' # True
'in' in 'violet' # False

#membership in list, tuple, set...
'in' in ['in', 'out']        # True
'in' in ['indigo', 'violet'] # False

#membership in dictionary is being one of the keys
'in' in {'in': 'out'} # True
'in' in {'out': 'in'} # False
```

- translates to `b.__contains__(a)` for objects

### For / While loops

```
nums = [1, 2, 3, 4, 5]

#break and continue:
for num in nums:
  if num == 3:
    print('found')
    # break
    continue # next loop iteration
  print(num)
```

```
#nested:
for num in nums:
  for letter in 'abc':
    print(num, letter)

#for range:
for i in range(10):
  print(i) # 0-9

for i in range(1, 11):
  print(i) # 1-10

#while:
x = 0
while x < 10:
  print(x) # 0-9
  x += 1

while True:
  if x == 5:
    break
  print(x)
  x += 1
```

- can use **break** or **continue**

## Functions

---

- uses the **def** keyword

```
def hello_func():
  # pass # no errors on empty function or object
  return 'Hello Function!'

print(hello_func)    # function location in memory
print(hello_func()) # executing function

#parameters:
def hello_func(greeting):
  return '{} Function.'.format(greeting)

print(hello_func('Hello'))

#default params / keyword args:
def hello_func(greeting, name='You'):
  return '{}, {}.'.format(greeting, name)

print(hello_func('Hello', name='Corey')) # 'Hello, Corey.'
print(hello_func('Hello', 'Corey'))      # 'Hello, Corey.'

#arbitrary args:
def student_info(*args, **kwargs):
  print(args)   # variable positional arguments
  print(kwargs) # variable keyword arguments

student_info('Math', 'Art', name='John', age=22)
#prints tuple of positional arguments
#prints dictionary of keyword arguments

#unpacking:
courses = ['Math', 'Art']
info = {'name': 'John', 'age': 22}

student_info(*courses, **info)
```

- Keyword Arguments
- parameters can be positional or keyword in a function call
    - required positional arguments must come before keyword arguments
    - if multiple keyword arguments, their order doesn't matter
- keyword arguments allow for:
    - leaving off arguments with default values

  - can rearrange arguments to make them readable
  - calling by name shows what arguments represent

Code example:

```
def quadratic(a, b, c):
  ...

quadratic(31, 42, 54)       # passing positional arguments
quadratic(a=31, b=42, c=54) # passing keyword arguments
quadratic(c=54, a=31, b=42) # order doesn't matter
quadratic(31, 42, c=62)     # can mix
values = (1,2,3)
quadratic(*values)          # unpacking with *
values = { 'a': 1, 'b': 2, 'c': 3 }
quadratic(**values)         # unpacking with **, only with dictionaries

#requiring arguments to be named
def product(*numbers, initial=1): # *numbers captures all positional args given
  total = initial
  for n in numbers:
    total *= n
  return total

product(4, 4)    # 16
prduct(4, 5, initial=2) # 40

def join(*iterables, joiner):
  ...

join([...],[...], joiner=0)
join([...],[...]) # error, no argument for joiner

#only accept keyword-only arguments with * alone

def render(..., ..., *, status=None, using=None,...)
def random_password(*, upper, lower, digits, length)


#arbitrary keyword arguments


def format_attributes(**attributes):
  ...
```

- star operator (*) can be used:
  - for unpacking arguments in function call
  - for variadic arguments in function parameters
    - * for positional
    - ** for keywords
    - can mix