**Python Cheat Sheet**

**JUST THE BASICS**

**CREATED BY: Arianne Colton and Sean Chen**

---

**GENERAL**

- Python is case sensitive
- Python index starts from 0
- Python uses whitespace (tabs or spaces) to indent code instead of using braces.

**HELP**

- Help Home Page: `help()`
- Function Help: `help(str.replace)`
- Module Help: `help(os)`

**MODULE (AKA LIBRARY)**

Python module is simply a *.py* file

- List Module Contents: `dir(module1)`
- Load Module: `import module1`
- Call Function from Module: `module1.func1()`

* import statement creates a new namespace and executes all the statements in the associated *.py* file within that namespace. If you want to load the module’s content into current namespace, use `from module1 import *`.

---

**SCALAR TYPES**

**Check data type**: `type(variable)`

**SIX COMMONLY USED DATA TYPES**

1. `int/long` - Large integer automatically converts to long
2. `float` - 64 bits, there is no ‘double’ type
3. `bool` - True or False
4. `str` - ASCII valued in Python 2.x and Unicode in Python 3
   - String can be in single/double/triple quotes
   - String is a sequence of characters, thus can be treated like other sequences
   - Special character can be done via \ or preface with `r`

```python
str = r‘this\?	\f\’
```

- String formatting can be done in a number of ways
  ```python
template = ‘%d,%d’;
str = template % (4,8, ‘hola’, 2)
```

---

**DATA STRUCTURES**

**Note**: All non-Get function call i.e. `list1.sort()` examples below are in-place (without creating a new object) operations unless noted otherwise.

### TUPLE

One dimensional, fixed-length, **immutable** sequence of Python objects of ANY type.

- **Notation**: `list1[start:stop]` (`if step is used`)  

- **Sequence types** include `str`, `array`, `tuple`, `list`, etc.

---

**DICTIONARY (HASH MAP)**

- *KeyError* exception if the key does not exist.
- **‘get()’** by default (aka no ‘default Value’) will return *None* if the key does not exist.
- **‘default Value’** will be returned.
- **‘keys()’** returns the keys of lists and values in the same order. However, the order is not any particular order, aka it is most likely not sorted.

Valid dict key types:
- Keys have to be immutable like scalar types (int, float, string) or tuples (all the objects in the tuple need to be immutable too)
- The technical term here is ‘hashability’, check whether an object is hashable with the `hash('this is string')`, `hash((1, 2))` - this would fail.

### SET

- A set is an unordered collection of UNIQUE elements.
- You can think of them like dicts but keys only.

- **Create Set**: `set([3, 6, 3]) or set([3, 6, 3])`
- Test Subset: `set1.issubset(set2)`
- Test Superset: `set1.issuperset(set2)`
- Test sets have same content: `set1 == set2`

### SLICING FOR SEQUENCE TYPES

- **WARNING**: bisect module functions do not check whether the list is sorted, doing so would be computationally expensive. Thus, using them in an unsorted list will succeed without error but may lead to incorrect results.

---

**SCALAR TYPES**

- `str()`, `bool()`, `int()` and `float()` are also explicit type cast functions.

5. NoneType(None) - Python `null` value (ONLY one instance of None object exists)
   - None is not a reserved keyword but rather a unique instance of ‘NoneType’
   - None is common default value for optional function arguments:

```python
def func(a, b, c = None):
    print(a, b, c)
```

- Common usage of None:

```python
if variable is None:
    print('none')
```

6. `datetime` - built-in python `datetime` module provides `datetime`, `date`, `time` types.
   - `datetime` combines information stored in `date` and `time`

Create datetime from String

```python
dt1 = datetime.strptime('20091031', '%Y%m%d')
```

Get ‘date’ object

```python
dt1.date()
```

Get ‘time’ object

```python
dt1.time()
```

Format datetime to String

```python
dt1.strftime('%Y/%m/%d')
```

Change Field Value

```python
dt2 = dt1.replace(minute = 0, second = 10)
```

Get Difference

```python
diff = dt1 - dt2
```

**Notation**: Most objects in Python are mutable except for ‘strings’ and ‘tuples’

---

**FUNCTIONS**

**Call Function from Module**

- `module1.func1()`

---

**APPENDIX**

**Module Help**

- `help(os)`

**Help Home Page**

- `help()`

**Load Module**

- `import module1`

**List Module Contents**

- `dir(module1)`

**Supplied Function**

- `dir()`

**Check Membership**

- `in`

**List**

One dimensional, **mutable** sequence of Python objects of ANY type.

Create List

```python
list1 = [1, 'a', 3]
```

Concateenate Lists

```python
list1 + list2
```

Append to End of List

```python
list1.append('b')
```

Insert to Specific Position

```python
list1.insert(posIdx, 'b')
```

Inverse of Insert

```python
valueAtIdx = list1.pop(posIdx)
```

Remove First Value from List

```python
list1.remove('a')
```

Check Membership

```python
if value in list1:
    print('in')
```

**Sort List**

```python
list1.sort()
```

**Sort with User-Supplied Function**

```python
list1.sort(key = lambda x: len(x))
```

**Inverse of Insert**

```python
list1.extend(list2)
```

**Slice List**

```python
list1[::2]
```

**Reverse a string**

```python
str1[::-1]
```
**FUNCTIONS**

Python is pass by reference, function arguments are passed by reference.

- **Basic Form:**
  ```python
def func1(posArg1, keywordArg1 = 1, ..):
```

**Note:**
- Keyword arguments MUST follow positional arguments.
- Python by default is NOT "lazy evaluation", expressions are evaluated immediately.

- **Function Call Mechanism:**
  1. All functions are local to the module level scope. See 'Module' section.
  2. Internally, arguments are packed into a tuple and dict, function receives a tuple 'args' and dict 'kwargs' and internally unpack.
- **Common usage of 'Functions' are objects**:
  ```python
def func1(ops = [str.strip, user_define_func, ..], ..):
    for function in ops:
      value = function(value)
```

**RETURN VALUES**

- **None** is returned if end of function is reached without encountering a return statement.
- **Multiple values return via ONE tuple object**

```
return (value1, value2) == value1, value2 = function(..)
```

**ANONYMOUS (AKA LAMBDA) FUNCTIONS**

- **What is Anonymous function?**
  A simple function consisting of a single statement.
  ```python
  lambda x : x * 2
  # def func1(x): return x * 2
  
  # Application of lambda functions : 'curring' aka deriving new functions from existing ones by partial application
  x60 = lambda x : pd.rolling_mean(x, 60)
  ```

**USEFUL FUNCTIONS (FOR DATA STRUCTURES)**

1. **Enumerate** returns a sequence (i, value) tuples
   where i is the index of current item.
   ```python
   for i, value in enumerate(collection):
   ```

   **Application:** Create a dict mapping of value (assumed to be unique) to their locations in the sequence.

2. **Sorted** returns a new sorted list from any sequence
   ```python
   sorted([2, 1, 3]) == [1, 2, 3]
   ```

**CONTROL AND FLOW**

1. **Operators for conditions in 'if else':**
   ```python
   var1 is var2
   ... are different object
   var1 = is not var2
   Check if two variables have same value
   ```

2. **Common usage of 'for' operator:**
   ```python
   iterating over a collection (i.e. list or tuple) or an iterator
   For element in
   ... if elements are sequences, can be 'unpacked'
   For a, b, c in iterator:
   ```

3. **pass** - no-op statement. Used in blocks where no action is to be taken.

4. **Ternary Expression** - aka less verbose 'if else'
   ```python
   Basic Form:
   value = true-exp if condition
   else false-exp
   ```

5. **No switch/case statement, use if/elif instead.**

**OBJECT-ORIENTED PROGRAMMING**

1. **'object'** is the root of all Python types
2. Everything (number, string, function, class, module, etc.) is an object, each object has a 'type'. Object variable is a pointer to its location in memory.
3. All objects are reference-counted.
   ```python
   sys.getrefcount(5) == x
   a = 5, b = a
   # This creates a 'reference' to the object on the right side of =, thus both a and b point to 5
   sys.getrefcount(5) == x + 1
   del(a); sys.getrefcount(5) == x + 1
   ```

4. **Class Basic Form:**
   ```python
   class MyObject(object):
   
   # 'self' is equivalent of 'this' in Java/C++
   def _init_(self, name):
     self.name = name
     def memberFunc1(self, arg1, ..):
     ...
     @staticmethod
     def classFunc2(arg1):
     ...
     obj1 = MyObject('name1')
     obj1.memberFunc1('a')
     MyObject.classFunc2('b')
   ```

5. **Useful interactive tool**
   ```python
   dir(variable1) # list all methods available on the object
   ```

**COMMON STRING OPERATIONS**

- **Concatenate List/Tuple with Separator**
  ```python
  str1 = 'My name is [0] (name)'
  newString1 = str1.format('Sean', name = 'Chen')
  ```

- **Format String**
  ```python
  string1 = 'My name is {0} (name)
  newString1 = string1.format('Sean', name = 'Chen')
  ```

- **Split String**
  ```python
  stringList1 = string.split(sep)
  ```

- **Get Substring**
  ```python
  start = 1; string1[start:8]
  ```

- **String Padding with Zeros**
  ```python
  month = '5';
  month.zfill(2) == '05'
  month = '12';
  month.zfill(2) == '12'
  ```

**EXCEPTION HANDLING**

1. **Basic Form:**
   ```python
   try:
     ...
   except ValueError as e:
     ...
   except (TypeError, AnotherError):
     ...
   finally:
     ... # clean up, e.g. close db
   ```

2. **Raise Exception Manually**
   ```python
   raise AssertionError # assertion failed
   raise SystemExit
   raise RuntimeError('Error message : ')
   ```

**LIST, SET AND DICT COMPREHENSIONS**

Syntactic sugar that makes code easier to read and write

1. **List comprehensions**
   - Concisely form a new list by filtering the elements of a collection and transforming the elements passing the filter in one concise expression.
   - **Basic form**:
     ```python
     [expr for val in collection if condition]
     ```
     A shortcut for:
     ```python
     result = []
     for val in collection:
       if condition:
         result.append(expr)
     ```
     The filter condition can be omitted, leaving only the expression.

2. **Dict Comprehension**
   - **Basic form**:
     ```python
     {key:expr: value:expr for value in collection if condition}
     ```
   - A shortcut for:
     ```python
     result = {}:
     for val in collection:
       if condition:
         result.update(expr)
     ```
     The filter condition can be omitted, leaving only the expression.

3. **Set Comprehension**
   - **Basic form:** same as List Comprehension except with curly braces instead of []
   - A shortcut for:
     ```python
     {expr for val in collection if condition}
     ```
     The filter condition can be omitted, leaving only the expression.

4. **Nested List Comprehensions**
   - **Basic form**:
     ```python
     [expr for val in collection if condition]
     ```
     A shortcut for:
     ```python
     result = []
     for val in collection:
       if condition:
         result.append(expr)
     ```
     The filter condition can be omitted, leaving only the expression.