Python cheat sheet

Comparison operators

Used to compare two objects. Returns `True` or `False`.

- `x < y` (less than)
- `x <= y` (less than or equal to)
- `x > y` (greater than)
- `x >= y` (greater than or equal to)
- `x == y` (is equal to)
- `x != y` (is not equal to)

These can be combined using `and` and `or`.

Example

```
x = 0
y = 5

if x < y:
    print("x is smaller than y")

if (x < y) and (x < 1):
    print("x is smaller than y and x is less than 1")
```

Conditionals

```
if  condition_one:
      do_something
elif  condition_two:
      do_something_else
elif  condition_three:
      do_another_thing
else:
      do_yet_another_thing
```

where `condition_one`, `condition_two` and `condition_three` are things that can be read as `True` or `False`. *else* is run if none of the other conditions are met. If multiple conditions are met, only the first one is executed.

Example

```
x = 2
if x < 0:
    print("x is negative")
elif x == 0:
    print("x is zero")
```

```
elif x > 0 and x < 100:
    print("x is positive but less than 100")
else:
    print("x is bigger than 100")
```

Loops

for syntax

```
for x in iterator:
    something_to_x
```

where `iterator` is something like `range(10)` or a list that has multiple entries.

Example .. sourcecode:: python

for i in range(10):

print(i)

while syntax

```
while condition:
    something
```

where `condition` is something that can be read as `True` or `False`

Example .. sourcecode:: python

i = 0 while i < 10:

print(i) i = i + 1

> ***Warning***
>
> If you forgot the `i = i + 1` line in the code above, it would create and infinite loop and your code would freeze. This is a common mistake when using `while` loops.

`continue` and `break` syntax

- `continue` hops to the next iteration of the loop
- `break` terminates the loop

Example

```
# Will print i from 6 to 90
i = 0
while i < 100:
```

```
    i = i + 1
    if i < 5:
        continue

    if i > 90:
        break

    print(i)
```

Datatypes

Single-value datatypes

- `int` (integer)
- `bool` (True or False)
- `float` (decimal number)

List-like objects

`list`

- **Specs**:
  - collection of arbitrary objects
  - indexed by number (starting from 0)
- **Creating new**:
  - `some_list = []` creates a new, empty list
  - `some_list = [1,2,3]` creates a new list with three entries
- **Adding new entry**:
  - `some_list.append(1)` appends the integer `1` to the end of the list
  - `some_list.append({})` appends an empty dictionary to the end of the list
- **Remove entry**:
  - `some_list.pop(1)` returns the second entry and removes it from the list
- **Getting values**:
  - `some_list[0]` gives first entry in list
  - `some_list[-1]` gives last entry in list
  - `some_list[1:3]` gives the second and third entry in list
- **Setting values**:
  - `some_list[0] = 5` sets the first value to `5`

- `some_list[-1] = 5` sets the last value to `5`
- `some_list[1:3] = ["test",8]` sets the second and third entries to `"test"` and `8`, respectively.

`tuple`

- **Specs**:
  - collection of arbitrary objects
  - behaves just like a list *except* that once it is created it cannot be modified.
- **Creating new**:
  - `some_tuple = (1,2,3)` creates a new tuple
- **Adding new entry**: can't be done
- **Remove entry**: can't be done
- **Getting values**:
  - Indexing and slicing rules just like lists
- **Setting values**: can't be done

`dict`

- **Specs**:
  - collection of arbitrary objects
  - objects are indexed by keys
  - keys can be almost any type *except* lists and dictionaries.
  - dictionaries are not ordered, meaning that if you loop through them more than once, the items could pop out in a different order
- **Creating new**:
  - `some_dict = {}` creates a new, empty dictionary
  - `some_dict = {"cows":27,18:"dogs"}` creates a new dictionary with `"cows"` keying to the value `27` and `18` keying to the value `"dogs"`
- **Adding new entry**:
  - `some_dict["meddling"] = "kids"` creates a key/value pair where the key `"meddling"` gives the value `"kids"`
- **Remove entry**:
  - `some_dict.pop("meddling")` would return `"kids"` and remove the `"meddling/kids"` key/value pair from the dictionary
- **Getting values**:
  - `some_dict["meddling"]` would return `"kids"`

    - `list(some_dict.keys())` returns list of keys
    - `list(some_dict.values())` returns list of values
    - `list(some_dict.items())` returns list of tuples with all key/value pairs
- **Setting values**:
    - `some_dict["scooby"] = "doo"` would key the value "doo" to the key `"scooby"`

`string`

- **Specs**:
    - stores text
    - behaves similarly to a list where every entry is a character
- **Creating new**:
    - `some_string = "test"` creates a new string storing test
    - Note that text in the string must have `"` around it.
- **Adding new entry**: can't be done
- **Removing entry**: can't be done
- **Getting values**: just like a list
    - `some_string[0]` returns the first letter
    - `some_string[-1]` returns the last letter
    - `some_string[1:3]` returns the second and third letter
- **Setting values**: just like a list
    - `some_string[0] = "c"` sets the first letter to `"c"`

`numpy.array`

- **Specs**:
    - collection of numerical objects of the same type
    - less flexible than a list (all objects must be same type, can't change dimensions after created).
    - collection of numpy functions allow extremely fast enumeration and access
    - requires `import numpy` at top of program
- **Creating**:
    - `numpy.zeros((10,10),dtype=int)` creates a new 10x10 integer array of zeros
    - `numpy.array([1.0,1.3,2.3],dtype=float)` creates a new 3 entry array of floats with input list values
- **Adding new entry**:

    - Can't really be done
    - `y = numpy.append(x,1.0)` will create a copy of *x* with 1.0 appended to it.
- **Removing entry**:
    - Can't really be done
    - `y = numpy.delete(x,0)` will create a copy of *y* with the first element removed.
- **Getting values**:
    - Extremely powerful (and sometimes complex)
    - `x[0]` returns the
    - `x[0,0,0]` returns the bottom left corner of a 3d array
    - `x[0:5]` returns the first five entries in a 1d array
    - `x[0,:]` returns the whole first column of a 2d array
    - `x[:,:,:,2]` returns a 3d slice at the third position on along the fourth dimension of a 4d array
- **Setting values**:
    - Exact same indexing and slicing rules as getting values

Libraries

(how to import and stuff)

important libraries math random numpy scipy matplotlib os combinations