Python Cheatsheet

Creating and assigning variables

```
# a number
x = 5
y = -10.6

# strings
hello = "Hi, Python!"

# booleans
pythonIsHard = False
pythonIsFun = True
```

Printing things

`print()` is a function that prints things to the console. Those things can either be strings, or variables (which Python will convert to strings)

```
print("hello")  # prints "hello"
print(hello)    # prints "Hi, Python!"
```

You can 'add' strings together ("concatenation") with `+`

```
city0 = "Evanston"
city1 = "Chicago"
cities = city0 + " and " + city1
print(cities)
```

This can be used to print *multiple* things to the console on the same line. But all those `+` can be hard to read, so you can also make "F-strings" which can contain Python

```
print("the best number is " + str(x) + ", but " + str(y) + " is good too")
print(f"the worst number is {x}, but {y} is bad too.")
print(f"....but {y + x + 1000} would be the best")
```

Asserts are powerful testing tools that let you stop the program if something is `False`, but will not do anything if it is `True`

```
assert city0 == city1, f"cities should be the same! {city0} is not the same
as {city1}"
```

Comparing things

You can compare things with `==`, `!=`, `<`, `>`, `<=`, `>=`. **Be careful to not use `=` by itself**, that assigns a new value!

```
notTheSame = x != y
stillNotTheSame = not x == y

if x > y:
    print(f"{x} is bigger than {y}")
elif x == y:
    print(f"{x} is the same as {y}")
else:
    print(f"{x} is less than {y}")
```

You can also compare strings. The same string will be equal but `<` or `>` will compare them *alphabetically*

```
if city0 > city1:
    print(f"{city0} is after {city1}")
elif city0 == city1:
    print(f"{city0} is the same as {city1}")
else:
    print(f"{city0} is before {city1}")
```

`and`, `or`, and `not` are also operators. If your statements get complex, add parentheses.

```
city0 != city1 or not x == y
(city0 != city1) or not (x == y)
```

Or even better, save each clause as a variable, that way you can test them independently:

```
citiesNotTheSame = city0 != city1
numbersNotTheSame = not (x == y)
allDifferent = citiesNotTheSame and citiesNotTheSame
print(f"citiesNotTheSame:{citiesNotTheSame}, numbersNotTheSame:
{numbersNotTheSame}")
print(f"allDifferent:{allDifferent}")
```

Lists

You can create an list explicitly, or use a range function to create an list of numbers that count up. These both create the same list.

```
arr0 = [0,1,2,3,4]
arr1 = range(0,5)
```

You can also create an list by multiplying a string and a number

```
allExes = "x"*15
print(f"allExes is: {allExes}")
```

will print:

```
allExes is: xxxxxxxxxxxxxxx
```

String comprehensions are special syntax in `[]` that create lists

```
squareList = [number*number for number in arr0]
evenSquareList = [x for x in squareList if (x%2==0)]

print(f"squareList is: {squareList}")
print(f"evenSquareList is: {evenSquareList}")
```

this will print

```
squareList is: [0, 1, 4, 9, 16]
evenSquareList is: [0, 4, 16]
```

You can “subscript” (that is, “look up data at a key”) with `[index]`. You can also set items with subscription. Negative numbers count from *back from the end* of this list, so are good for asking for the last element. Asking for an index that is outside the list (ie. the 5th element in a 5 element list). The code below will print `water`, `love`, `skittles`, then give an `IndexError` error.

```
elements = ["water", "air", "fire", "earth", "love"]
print(f"The first element is {elements[0]}")
print(f"The final element is {elements[-1]}")

elements[0] = "skittles"
print(f"The first element is {elements[0]}")
```

You can also use subscript notation to make *new* lists from an existing list

```
print(f"The middle elements are: {elements[1:3]}")
print(f"All but the first elements: {elements[0:-1]}")
```

If you try to use subscription notation on a **non-list** Python will let you know with a `subscription` error.

You can add an element to the end of alist with `myList.append(someElement)`, add at an index with `myList.insert(index,someElement)`, remove the last element with `mylist.pop()`, or clear it with `myList.clear()`. Length is a bit different `len(myList)`

```
laughter = []
while len(laughter) < 4:
    laughter.append("ha")
    print(laughter)
```

You can also make Tuples, which are a lot like lists, but you can’t modify them after you make them. This code will cause a `'tuple' object does not support item assignment` error.

```
planets = ("jupiter", "saturn", "uranus", "neptune", "pluto")
planets[5] = "xena"
```

Loops

In Python, `for` loops iterate over every element of an list. So you need an list to pass them. You can create a new one with `range()`

```
for i in range(5):
    print (f"The square of {i} is {i*i}")
```

Or you can use an existing list variable

```
for number in evenSquareList:
    print (f"{number} is an even square")
```

`while` loops execute while the condition evaluates to `True`

```
count = 5
while count > 0:
    count = count - 1
    print(f"countdown: {count}")
print("*boom!*")
```

```
# run forever....
count = 5
while count > 0:
    count = count + 1
    print(f"never gonna: {count}")
print("give you up!")
```

If your condition is never `False`, they will run **forever**, and you have to hit CMD-C or CTRL-C to escape the process.

Functions and importing

Importing libraries is easy, just `import libraryName` or `from libraryName import someSmallPartOfLibrary` for just a part of a library. The most common libraries to import is `math` with gives you common math operations and `sys` which lets you do file system operations.

Functions are defined with `def` and have parameters in `()`. Make sure you have `def` `()` and `:` and indent the function body!

```
import math
def getDistance(x, y, z):
    sum = x*x + y*y + z*z
    return math.sqrt(sum)

vx = 5
vy = 10
vz = 15
dist = getDistance(vx, vy, vz)
print(f"The length of a vector {vx} {vy} {vz} is {dist}, about
{round(dist)}")
```

Dictionaries

Dictionaries are lookup tables from string keys to some values, which may be numbers, strings, lists or even other dictionaries. Note that keys are *always* in `""`, and *every line but the last* needs a `,`.

```
translations = {
    "cat": {
        "spanish": "gato",
        "german": "katze",
        "finnish": "kissa",
        "yoruba": "ologbo"},
    "dog": {
        "afrikaans": "hond",
        "gujarati": "kutto",
        "hawaiian": "ilio"
    }
}
```

You can “subscript” (that is, “look up data at a key”)

```
catDictionary = translations["cat"]
```

You can also ask for an list of all the keys a dictionary has, and use that in a `for` loop.

```
catLanguages = catDictionary.keys()

for lang in catLanguages:
    print(f"'cat' in {lang} is '{catDictionary[lang]}' ")
```

`key in list` will return `True` or `False` depending if the key is in the dictionary or not. Asking for a key that is *not* in the dictionary will give you an error, so this is helpful!

```
canTranslateFish = "fish" in translations
print(f"Can I translate 'fish' with this dictionary? {canTranslateFish}")

if "french" in catLanguages:
    print(f"'A French cat is '{catDictionary['french']}' ")
else:
    print("No cat translation for French")
```

Classes

Classes create objects that have their own individual data, but also access to class *methods*, special functions that can run with `self.myMethodName()`

```
class Cat:
    catWord = "meow"

    def __init__(self, name):
        self.fish = 0
        self.mood = "happy"
        self.name = name

    def speak(self):
        return f"{self.name} says '{self.catWord}'"
```

`print(cat0)` will print:

```
<__main__.Cat object at 0x7fd7d4b3af40>
```

Not so helpful! So we can add a `__str__` method that turns it into a more useful string.

```
    def __str__(self):
        return f"{self.name}, a {self.mood} cat"
```

```
Bustopher, a happy cat
Grizabella, a sad cat
```