Python cheat sheet:

Operatoren:

De standaard operatoren voor wiskundige bewerkingen (+,-,*,/,**) worden als vanzelfsprekend ondersteld. Voor **int**egers en **float**ing point getallen doen deze functies wat je vanuit een wiskundig standpunt kan verwachten.

Op **str**ing en **list** objecten zijn enkel de operatoren + en * gedefinieerd.
'abc' of [1,2,3] kunnen beide vermenigvuldigd worden met een **int** (bijvoorbeeld 3) en het resultaat van die operatie is dan 'abcabcabc' or [1,2,3,1,2,3,1,2,3], respectievelijk.
De + operator voor deze objecten is gedefinieerd voor **str**+**str** of **list**+**list** en doet ook wat je zou verwachten: namelijk de 2 objecten die worden opgegeven aan elkaar plakken:
'abc'+'def' -> 'abcdef' of [1,2,3]+[4,5,6] -> [1,2,3,4,5,6].

Een fout die vaak gemaakt wordt is de + operator gebruiken met objecten van verschillend type. Bijvoorbeeld:

```
leeftijd = 18 # dit is een integer
print('Jouw leeftijd is ' + leeftijd + ' jaar.')
```

Hier wordt er geprobeerd om strings en integers met elkaar op te tellen. De variabele leeftijd moet eerst omgezet worden naar een string:

```
print('Jouw leeftijd is ' + str(leeftijd) + ' jaar.')
```

Hetzelfde voor lijsten:

```
mijnlijst = [1,2,3,4]
print(mijnlijst + 5)
```

Aangezien 5 een integer is zal Python klagen wanneer we deze proberen op te tellen bij een lijst. De juiste manier hier is om te zeggen dat we de lijst met het element 5 willen optellen bij mijnlijst:

```
print(mijnlijst + [5])
```

Built-in functies:

Er zijn in Python al veel functies voorzien zodat je niet telkens het wiel moet heruitvinden om eenvoudige problemen op te lossen. Op de officiële documentatie website (https://docs.python.org/2/library/functions.html) vind je ze allemaal met uitleg en voorbeelden.

Enkele handige om te onthouden zijn:

len(object) -> integer	Geeft de lengte van een object (string, lijst, tupel) terug als integer
range(start,eind,stap) -> list	Geeft een lijst terug met gehele getallen van start tot eind (zonder eind zelf) met een stapgrootte stap.
abs(getal) -> getal	Geeft de absolute waarde van een getal terug
int(x), float(x), str(x), bool(x)	Zet x om in het gevraagde type en geeft dit resultaat terug.
raw_input([prompt]) -> string	Schrijft de tekst in prompt naar de shell en wacht tot de gebruiker een input geeft. Deze input wordt als string teruggegeven.

Indexering en slicing:

Objecten zoals tupels, lijsten en strings bestaan uit verschillende elementen. De individuele elementen in zulke objecten kunnen aangesproken worden door hun index. Geldige indexen gaan van 0 tot en met de lengte van het object min 1.

Bijvoorbeeld:

```
mijnlijst = [1,2,3,4,5]
print(mijnlijst[1]) # dit print de waarde 2 uit, omdat op index 1 in mijnlijst de waarde 2 staat
mijnlijst[1] = 0
print(mijnlijst[1]) # dit print nu de waarde 0, omdat de lijst in de vorige regel gewijzigd werd
```

Strings werken gelijkaardig:

```
mijnstring = 'dit is een test'
print(mijnstring[1]) # dit print het karakter 'i' uit
mijnstring[1] = 'z' # dit zal een error geven, strings en tupels kunnen niet aangepast worden
```

In plaats van slechts 1 index per keer aan te spreken kunnen ook meerdere elementen tegelijk aangesproken worden door gebruik te maken slicing: object[begin:eind].

Bijvoorbeeld:

```
mijnlijst = [0,1,2,3,4,5,6,7,8,9]
print(mijnlijst[2:6]) # dit print [2,3,4,5] uit
mijnlijst[2:6] = []
print(mijnlijst) # dit print [0,1,6,7,8,9] uit
```

Voor strings werkt slicing ook, maar net zoals met enkelvoudige indexering kunnen strings niet aangepast worden.

Verder hoeven begin en eind ook niet per se opgegeven te worden:

```
mijnlijst = [0,1,2,3,4,5,6,7,8,9]
print(mijnlijst[2:]) # dit print de lijst vanaf index 2 tot het einde uit
print(mijnlijst[:4]) # dit print de lijst vanaf het begin tot en met index 3 uit
```

Slicing – vervolg:

Slicing zonder begin en einde maakt een kopie van de lijst aan. Hieronder een voorbeeld waarom dit handig is. Stel dat we een functie willen schrijven die het kleinste element in een lijst teruggeeft.

```
def vind_kleinste_1(L):
    L.sort()
    return L[0]

mijnlijst = [5,2,4,1,3]
print vind_kleinste_1(mijnlijst) # hier wordt correct de waarde 1 geprint
print(mijnlijst) # hier wordt [1,2,3,4,5] geprint, de originele lijst is dus gesorteerd
```

Soms willen we echter niet dat de lijst die we aan de functie meegeven wordt aangepast. We kunnen onze functie anders schrijven (bijvoorbeeld met een loop), maar stel dat we het toch met sort() willen doen, dan moeten we op een kopie van de lijst werken:

```
def vind_kleinste_2(L):
    tijdelijke_lijst = L
    tijdelijke_lijst.sort()
    return tijdelijke_lijst[0]

mijnlijst = [5,2,4,1,3]
print(vind_kleinste_2(mijnlijst)) # hier wordt correct de waarde 1 geprint
print(mijnlijst) # hier wordt nog steeds [1,2,3,4,5] geprint, de originele lijst is dus weer gesorteerd
```

Het probleem is dat tijdelijke_lijst eigenlijk dezelfde lijst is als L (dus de lijst mijnlijst in het voorbeeld). Als we willen dat tijdelijke_lijst een nieuwe lijst is die een kopie is van L, dan moeten we slicing gebruiken:

```
def vind_kleinste_3(L):
    tijdelijke_lijst = L[:] # merk op dat we een kopie van L maken
    tijdelijke_lijst.sort()
    return tijdelijke_lijst[0]

mijnlijst = [5,2,4,1,3]
print(vind_kleinste_3(mijnlijst)) # hier wordt correct de waarde 1 geprint
print(mijnlijst) # hier wordt correct [5,2,4,1,3] geprint, dus de originele lijst is niet veranderd
```

Iteraties:

Een **while** loop herhaalt een blok code zolang de loopvoorwaarde True is. Bijvoorbeeld:

```
invoer = ''
while invoer != 'q':
    invoer = input("Geef de letter 'q' in: ")
```

Dit programma zal de gebruiker vragen om iets in te voeren. Zolang de gebruiker niet de letter 'q' invoert zal het programma de vraag herhalen.

Een **for** loop in Python gaat stap voor stap door alle elementen in een 'itereerbaar object'. Dit object kan een string, lijst of tupel zijn, maar bijvoorbeeld ook een bestand. Bijvoorbeeld:

```
mijnlijst = [1,2,3,4,5,6,7,8]
for elk_element in mijnlijst:
    print(elk_element)
```

Dit programma creëert een lijst, gaat met een for loop door alle elementen in de lijst en print elk element dat het tegenkomt uit naar de shell. *elk_element* is een iteratie variabele. De for loop begint aan het begin van de lijst, zet het eerst object van de lijst tijdelijk in de variabele elk_element en voert de loop-code uit. (In dit geval **print**(elk_element).) Na het uitvoeren van deze code zet de for het tweede element in elk_element en wordt de code weer uitgevoerd. Enzovoort tot alle elementen in mijnlijst geprint zijn.

De functie range(begin,eind,stap) geeft een 'generator' terug die begint op het getal *begin* en in stappen met grootte *stap* naar eind gaat. (Het getal eind zelf zit niet meer in de lijst.) Dit is handig omdat we dan het stukje voorbeeldcode van hierboven veel korter hadden kunnen schrijven als:

```
for elk_element in range(1,9,1):
    print(elk_element)
```

De stapgrootte moet niet steeds gegeven worden, als die niet gegeven is dan wordt een stapgrootte van 1 ondersteld. De range functie kan ook aangeroepen worden met slechts 1 argument en in dat geval wordt de lijst van 0 tot de gegeven waarde teruggegeven. Bv: range(5) -> [0,1,2,3,4]

Merk op: dit geeft in combinatie met de len() functie een tweede manier om elementen in een lijst met een for loop één voor één aan te spreken:

```
mijnlijst = ['testlijst', [1, 2], -5, True]
for i in range(len(mijnlijst)):
    print(mijnlijst[i])
```

Om te begrijpen waarom dit nuttig is, voer de volgende twee programma's eens uit:

```
mijnlijst = ['testlijst', [1, 2], -5, True]
for element in mijnlijst:
    element = 0
print(mijnlijst)
```

```
mijnlijst = ['testlijst', [1, 2], -5, True]
for i in range(len(mijnlijst)):
    mijnlijst[i] = 0
print(mijnlijst)
```

Methodes op strings:

Alle methods met uitleg kunnen opgevraagd worden door help(**str**) in te voeren in de shell. In de onderstaande tabel vind je enkele interessante om te onthouden:
Functie argumenten die tussen [ ] staan zijn optioneel.

find(sub [,start [,end]]) -> int	Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation. vb: 'dit is een test'.find('is') -> 4
isdigit() –> bool	Return True if all characters in S are digits and there is at least one character in S, False otherwise.
islower() / isupper() -> bool	Return True if all cased characters in S are lowercase/uppercase and there is at least one cased character in S, False otherwise.
lower() / upper() -> string	Return a copy of the string S converted to lowercase / uppercase.
join(iteratable) -> string	Return a string which is the concatenation of the strings in the iterable. The separator between elements is S. vb: " ".join(["dit,"is","een","test"]) -> "dit is een test"
split([sep [,maxsplit]]) -> list of strings	Return a list of the words in the string S, using sep as the delimiter string. If maxsplit is given, at most maxsplit splits are done. If sep is not specified or is None, any whitespace string is a separator and empty strings are removed from the result. 'dit is een test'.split() -> ['dit','is','een','test'] 'dit is een test'.split(' ', 1) -> ['dit','is een test'] '1;2;3;4'.split(',') -> ['1','2','3','4']
strip([chars]) -> string	Return a copy of the string S with leading and trailing whitespace removed. If chars is given and not None, remove characters in chars instead. vb: ' test\n\n '.strip() -> 'test' vb: 'xxxxDit is een testx'.strip('x') -> 'Dit is een test'
replace(old,new [,count]) -> string	Return a copy of string S with all occurrences of substring old replaced by new. If the optional argument count is given, only the first count occurrences are replaced. vb: 'dit is een test'.replace('is','was') -> 'dit was een test' vb: 'blablabla'.replace('bla','',1)-> 'blabla'

Methodes op lijsten:

Alle methods met uitleg kunnen opgevraagd worden door help(**list**) in te voeren in de shell. In de onderstaande tabel vind je enkele interessante om te onthouden:

Functie argumenten die tussen [ ] staan zijn optioneel.

Veel methodes op lijsten geven niets terug, maar passen de lijst aan.

print mylist.sort() zal bij uitvoering de lijst *mylist* sorteren, maar aangezien de sort() functie geen return waarde heeft zal er *None* geprint worden. Uitdrukkingen als *mylist = mylist.sort()* zijn dus gevaarlijk aangezien er op het einde *None* in de variabele *mylist* zal staan.

append(object)	Append object to end
	l = [1,2,3]
	l.append(4)
	# l is now [1,2,3,4]
	l.append([5,6,7])
	# l is now [1,2,3,4,[5,6,7]]

extend(iterable)	Extend list by appending elements from the iterable l = [1,2,3,4] l.extend([5,6,7]) # l is now [1,2,3,4,5,6,7]
index(value [,start [,stop]]) -> integer	Returns the first index at which value can be found, checking only the elements from start to end in the list.
insert(index, object)	Inserts object before index l = [1,2,3,5] l.insert(3,4) # l is now [1,2,3,4,5]
pop([index]) -> item	Removes the item at index from the list and returns it
reverse()	Reverses list
sort()	Sorts the list