Python Cheat Sheet

Python Cheat Sheet

Matematiska operatorer

Operator	Operation	Exempel
`**`	Upphöjt till	`3 ** 2 # Resultat: 9`
`%`	Modulo	`9 % 2 # Resultat: 1`
`//`	Heltalsdivision	`9  // 2 # Resultat: 4`
`/`	Division	`9 / 2 # Resultat: 4.5`
`*`	Multiplikation	`2 * 3 # Resultat: 6`
`-`	Subtraktion	`2 - 3 # Resultat: -1`
`+`	Addition	`2 + 3 # Resultat: 5`

Operatorerna ovan är listade i fallande prioriteringsordning.

Variabler och datatyper

Inbyggda datatyper

En variabel är ett namn med ett värde knutet till sig. En variabel är av en specifik *datatyp*. Variabler *tilldelas* ett värde enligt följande exempel (datatyp inom parentes):

```
pris    = 10                # Datatyp int (Heltal), utan decimalpunkt
sträcka = 4.5               # Datayp float (flyttal), med decimalpunkt
namn    = "Ada"             # Datatyp str (Sträng), teckenföljd
is_game_on = True           # Datatyp bool (Boolesk variabel), kan anta värdena True och Fals
even5   = [2, 4, 6, 8, 10]  # Datatyp list (Lista)
tabell = {"a":1, "b":2}     # Datatyp dict (Uppslagstabell)
```

En variabel kan skrivas över med t ex `pris += 0.5` (nu kommer `pris` att ha värdet `10.5`, och således också ha bytt datatyp till `float`). Ett annat exempel är `namn += " Lovelace"`. Nu kommer `namn` att utgöras av strängen `"Ada Lovelace"`.

Av ovanstående datatyper så är det enbart `list` och `dict` som är *muterbara*, dvs att deras innehåll kan ändras. Operationer på övriga variabeltyper skapar en ny variabel, eventuellt med samma namn, och nytt innehåll.

Vissa datatyper går att konvertera till andra datatyper. T ex ger `str(3.14)` resultatet `'3.14'` (konvertering från `float` till `str`) och `int("9")` ger resultatet `9` (konvertering från `str` till `int`).

`print`-satsen

För att skriva ut något till konsolen kan en `print`-sats användas. Exempel:

```
print("Hello, world") # Skriver ut: Hello, world
favorite_number = 42
print(favorite_number) # Skriver ut: 42
print(f"Mitt favorittal är {favorite_number}, förstås.")
# Skriver ut: Mitt favorittal är 42, förstås.
```

```
Hello, world
42
Mitt favorittal är 42, förstås.
```

`input`-satsen

Ett program kan vänta på indata från konsolen genom funktionen `input`. Denna funktion returnerar alltid en sträng (`str`). Exempel:

```
given_name = input("Ange ditt förnamn -> ")
print(f"Hej, {given_name}!")
```

Jämförelseoperatorer

Följande tilldelningar görs:

```
a = -2
b = a
c = 0
```

Nu kan t ex följande jämförelser göras:

Operator	Operation	Resultat
Lika med	`a == b`	`True`
Skilt från	`a != b`	`False`
Större än	`a > b`	`False`
Mindre än	`a < c`	`True`
Större än eller lika med	`a >= b`	`True`
Mindre än eller lika med	`a <= c`	`True`

Det går även att jämföra flera variabler i en följd, t ex `a == b < c`, vilket returnar `True`.

Logiska operatorer

Det finns tre inbyggda logiska operatorer i Python: `and`, `or` och `not`. Följande tilldelningar görs: `a = True`, `b = False`, `c = True` och `d = False`.

```
a = True
b = False
c = True
d = False
```

Operator	Operation	Resultat
`and`	`a and c`	`True`
`and`	`a and b`	`False`
`or`	`a or b`	`True`
`or`	`b or d`	`False`
`not`	`not a`	`False`
`not`	`not b`	`True`

Det går att tillämpa flera logiska operatorer i en följd, t ex `(a and not b) or (a and d)`, vilket returnerar `True`.

Villkor

I Python prövas ett *villkor* med `if`-satser. Syntaxen är

```
if <booleskt_uttryck_1>:
    <gör något om booleskt_uttryck_1 är True>
elif <booleskt_uttryck_2>:
    <Om inte booleskt_uttryck_1 var True, gör något annat om booleskt_uttryck_2
    är True>
else:
    <Gör något annat om inget av de tidigare prövade booleska uttrycken var True>
```

Detta kan exemplifieras i följande program:

```
my_number = 42
if my_number > 100:
    print("Talet är större än hundra.")
elif my_number >= 0:
    print("Talet är större än eller lika med noll")
else:
    print("Talet är negativt")
# Programmet skriver ut: Talet är större än eller lika med noll.
```

```
Talet är större än eller lika med noll
```

Loopar

`while`-loopar

I en loop kan kod processas flera gånger. Exempel:

```
a  = 0
while a < 5:
    print(a, end=" ")
    a += 1
# Skriver ut 0 1 2 3 4
```

```
0 1 2 3 4
```

Det går att kontrollera hur loopen körs genom nyckelorden `break` (avbryter aktuell loop) och `continue` (startar om loopen från början). Exempel:

```python
a = 0
while True:
    if a == 5:
        a = 7
        continue
    if a == 9:
        break
    print(a, end=" ")
    a += 1
# Skriver ut: 0 1 2 3 4 7 8
```

```
0 1 2 3 4 7 8
```

`for`-loopar

Se avsnitt Listor, underavsnitt Loopa genom listor. Nyckelorden `break` och `continue` fungerar på samma sätt i `for`-loopar som i `while`-loopar.

Dataföljder

Listor

En lista samlar olika värden eller objekt i en enda variabel. Vart och ett av dessa nås med ett *index*.

Vi deklarerar:

```python
lst = [10, 20, 30, 40, 50]
```

Åtkomst av element samt delar av listor

Adresserat index	Kod	Resultat
`0`	`lst[0]`	`10`
`1`	`lst[1]`	`20`
`-1`	`lst[-1]`	`50`
`-2`	`lst[-2]`	`40`
`2, 3`	`lst[2:4]`	`[30, 40]`
`0, 1, 2, 3`	`lst[:4]`	`[10, 20, 30, 40]`
`2, 3, 4`	`lst[2:]`	`[30, 40, 50]`
`1, 3`	`lst[1:4:2]`	`[20, 40]`

Adresserat index	Kod	Resultat

Kommentarer: Listans första element har index `0` och dess sista element har index `-1`. Med kolon-tecknet kan man få ut en del av en lista, t ex en lista med elementen med index `1, 2, 3` erhålls med `lst[1:4]` (till, men inte till och med, index 4). Det finns även en speciell notation som ger hela listan i omvänd ordning: `lst[::-1]` ger `[50, 40, 30, 20, 10]`.

En lista som innehåller t ex vartannat index av en given lista kan skapas med `lst[1:4:2]` (se tabellen ovan). Den innehåller vartannat index från `1` till (men inte till och med) index 4

Skapa listor

En lista kan skapas med den inbyggda funktionen `range`. T ex `long_list = list(range(100, 1000, 2))`. Detta kommer att skapa listan `[100, 102, 104,..., 998]`.

Funktionen `range` returnerar inte en lista, den returnerar ett itererbart objekt. Detta objekt kan inte skrivas ut direkt, men det går att göra om till en lista med funktionen `list`.

Det går även att skapa s.k *listomfattningar* enligt `lst = [2*i for i in range(5)]`, det ger listan `[0, 2, 4, 6, 8]`.

Antal element i en lista

Antalet element i en lista erhålls med den inbyggda funktionen `len`. Exempel:

```
long_list = list(range(100, 1000, 2))
print(len(long_list)) # Skriver ut antalet element: 450
```

Kontrollera om ett element finns i en lista

Förekomsten av ett element i en lista kan kontrolleras enligt följande:

```
long_list = list(range(100, 1000, 2))
500 in long_list # Returnerar True
501 in long_list # Returnerar False
```

Loopa genom listor

En lista kan gås igenom element för element i en `for`-loop enligt fäljande:

```
lst = list(range(5, 0, -1)) # Skapar listan [5, 4, 3, 2, 1]
for i in lst:
    print(i, end=" ")
# Matar ut: 5 4 3 2 1
```

Det behöver inte vara en lista som loopen iterar över. Det går t ex att loopa direkt över det objekt som `range` returnerar.

Lägga till och ta bort element från en lista

Följande exempel visar några tilläggs- och borttagningsoperationer:

```
lst = [10, 20, 30, 40]
lst.append(50) # lst är nu [10, 20, 30, 40, 50]
lst.remove(30) # lst är nu [10, 20, 40, 50]
del(lst[1])    # lst är nu [10, 40, 50]
lst.pop()      # Returnerar 50, lst är nu [10, 40]
a = lst.pop()  # a är nu 40, lst är nu [10]
```

Sortering av en lista

Följande exempel visar hur en lista kan sorteras

```
lst = [5, 3, 10, -1, 8]
sorted(lst) # Returnerar [-1, 3, 5, 8, 10]; lst är fortfarande [5, 3, 10, -1, 8]
lst.sort()  # Returnerar inget, lst är ändrad till [-1, 3, 5, 8, 10]
lst.sort(reverse=True) # lst är nu [10, 8, 5, 3, -1]
```

Uppslagstabeller

En uppslagstabell fungerar som en lista, men istället för numrerade index så har den `nyckel:värde`-par. Nyckeln kan vara vara av vilken immuterbar, datatyp som helst (t ex en sträng eller ett tal), värdet kan vara av vilken datatyp som helst, inklusive egendefinierade datatyper. Exempel:

```
my_dict = {"namn": "Anna",
           "ålder": 25,
           "längd": 1.75}

# Loopar genom uppslagstabellen
```

```python
for key, value in my_dict.items():
    print(f"{key}: {value}", end=", ")
    # Skriver ut "namn: Anna, ålder:25, längd:1.75, "

# Lägger till en ny nyckel med värde i form av en lista
my_dict["värden"] = [1, 2, 3]

# Raderar nyckeln "längd"
del my_dict["längd"]

# Alternativ loop genom uppslagstabellens nycklar
for key in my_dict.keys():
    print(f"{key}: {my_dict[key]}")
    # Skriver ut:
    # namn: Anna
    # ålder: 25
    # värden: [1, 2, 3]

# Kontroll av förekomsten av en nyckel:
"värden" in my_dict # Returnerar True
"längd" in my_dict  # Returnerar False
```

Strängar och strängmetoder

Strängar kontra listor

En sträng (datatyp `str`) kan i flera flera fall hanteras som en lista.

Vi deklarerar

```python
str1 = "ABC 123"
```

Nu gäller att `str1[1]` är `"B"` och `str1[-1]` är `"3"`. Däremot går det inte att utföra `str1[-1] = 4`; det ger ett felmeddelande eftersom strängar inte är muterbara. Antalet tecken i en sträng erhålls på samma sätt som antalet element i en lista med funktionen `len`; t ex ger `len(str1)` returvärdet `7`.

Delsträngar fungerar på samma sätt som att komma åt delar av en lista; se rubriken Listor, underrubrik Åkomst av element ovan.

Justerade strängar med text

En sträng kan högerjusteras eller centreras inom en given *fältbredd*. Exempel: Strängen `"ABC"` justeras så att den tar tio tecken i anspråk och högerjusteras.

`"ABC".rjust(10)` ger utskriften `"       ABC"`

Samma sträng, men centrerad i fältet:

`"ABC".center(10)` ger utskriften `"   ABC    "`

Samma sträng, men vänsterjusterad i fältet:

`"ABC".ljust(10)` ger utskriften `"ABC       "`

För att det ska bli någon effekt av justeringen måste fältbredden vara större än antalet tecken i strängen som skrivs ut.

Om f-strängar används blir syntaxen:

`f"{'ABC':>10}"` som ger `"       ABC"`

`f"{'ABC':^10}"` som ger `"   ABC    "`

`f"{'ABC':<10}"` som ger `"ABC       "`

Justerade strängar med tal

För att presentera ett värde med decimaler så kan avrundning göras inne i strängen.

```
pi = 3.14159265358979
print(f"Avrundat pi: {pi:.4f}")    # Fyra decimaler
print(f"Avrundat pi: {pi:10.4f}")  # Fyra decimaler med fältbredden 10
print(f"Avrundat pi: {pi:010.4f}") # Utfyllnadsnollor
```

```
Avrundat pi: 3.1416
Avrundat pi:     3.1416
Avrundat pi: 00003.1416
```

Vi ser att avrundning har skett korrekt på fjärde decimalen.

Strängmetoder

I följande exempel är följande deklaration gjord:

```
str1 = "Detta är en sträng som består av 42 tecken"
```

Metod	Exempel	Resultat	Kommentar
`.count()`	`str1.count("e")`	`5`	Ger antalet tecken av den specificerade sorten. Det går även att söka på en följd av tecken, då måste hela följden matchas för att räknas
`.upper()`	`str1.upper()`	`"DETTA ÄR EN STRÄNG SOM BESTÅR AV 42 TECKEN"`	Returnerar strängen där de alfanumeriska tecknen är versaler. `str1` är oförändrad.
`.lower()`	`str1.lower()`	`"detta är en sträng som består av 42 tecken"`	Returnerar strängen där de alfanumeriska tecknen är gemener. `str1` är oförändrad.
`.title()`	`str1.title()`	`"Detta Är En Sträng Som Består Av 42 Tecken"`	Konverterar det första tecknet i varje ord till versal. `str1` är oförändrad.
`.strip()`	`str1.strip()`	`"Detta är en sträng som består av 42 tecken"`	Tar bort mellanslag i början och i slutet av strängen (finns inga sådana i den aktuella strängen). `str1` är oförändrad.
`.replace()`	`str1.replace("42", "många")`	`"Detta är en sträng som består av många tecken"`)	Ersätter ett tecken, eller en följd av tecken, med andra tecken. `str1` är oförändrad.
`.split()`	`str1.split()`	`"['Detta', 'är', 'en', 'sträng', 'som', 'består', 'av', '42', 'tecken']"`	Gör om strängen till ord i en lista. Separatortecknet är mellanslag som standard. `str1` är oförändrad.

Metod	Exempel	Resultat	Kommentar
`.index()`	`str1.index("a")`	`4`	Returnerar första positionen av ett tecken eller en teckenföljd. `str1` är oförändrad.
`.isalnum()`	`"abc123".isalnum()`	`True`	Returnerar `True` om alla tecken som kontrolleras är alfanumeriska, dvs bokstäver eller siffror; annars `False`.
`.isalpha()`	`"abc123".isalpha()`	`False`	Returnerar `True` om alla tecken som kontrolleras är bokstäver, annars `False`.
`.isdigit()`	`"42".isdigit()`	`True`	Returnerar `True` om alla tecken som kontrolleras är siffror, annars `False`.

Att räkna det totala antalet tecken som finns i en sträng är inte en metod, det är en funktion:

```
num_chars = len(str1)
print(f"Strängen innehåller {num_chars} tecken.")
```

```
Strängen innehåller 42 tecken.
```

Funktioner

En funktion i är ett kodblock som kan anropas och processas som en separat del i programmet. Den kan, men behöver inte, returnera ett explicit värde (tal, sträng, lista,...). Om funktionen inte har en `return`-sats så kommer `None` att returneras.

Exempel på funktion utan returvärde

```python
def greet(name):
    print(f"Hej {name}!")

greet("Alice") # Funktionen körs, och skriver ut Hej Alice!
```

```
Hej Alice!
```

Denna funktion returnerar alltså `None`, vilket inte syns om resultet av funktionen inte skrivs ut.

Exempel på funktion med ett returvärde

```python
def divide(divisor, dividend):
    if dividend == 0:
        return None
    else:
        return divisor / dividend

a = divide(10, 5) # a är nu 2.0
a = divide(10, 0) # a är nu None
```

Exempel på funktion med flera returvärden

```python
pi = 3.14
def circle(radius):
    circ = pi * radius * 2
    area = pi * radius ** 2
    return (circ, area)

circ, area = circle(1)
print(f"Omkrets: {circ}, Area: {area}")
```

```
Omkrets: 6.28, Area: 3.14
```

Egendefinierade datatyper

En egendefinierad datatyp skapas med nyckelorden `class`. Den kan skapas och användas enligt följande:

```
# Datatypen skapas
class Car:
    '''
    Datatyp som samlar information om en bil.
    '''
    def __init__(self, brand, model_year):
        self.brand = brand
        self.model_year = model_year

# En instans skapas
a_bil = Car("Volvo", 2021)

# Informationen kan användas enligt
print(f"Variabeln a_bil är en bil av märket {a_bil.brand} ", end="")
print(f"med årsmodell {a_bil.model_year}")
```

```
Variabeln a_bil är en bil av märket Volvo med årsmodell 2021
```

Objekt av egendefinierade datatyper är muterbara, dvs de kan ändras.

Slumptal

Funktioner som har med slumptal att göra finns i en separat *modul*, nämligen `random`.

Exempel på slumpmässigt heltal

```
import random as rand
random_int = rand.randint(0, 99)
# random_int kommer nu att innehålla ett slumptal mellan, och inklusive, 0 och 99
```

Exempel på slumpmässigt flyttal

```
import random as rand
random_float = rand.random()
# random_float kommer nu att innehålla ett slumpmässigt flyttal mellan 0 och 1 (dock ej inklu
```

Exempel på att blanda en lista

```
import random as rand
lst = list(range(0, 10))
rand.shuffle(lst) # lst kommer nu att vara blandad
```

Exempel på att plocka ut ett slumpmässigt tal från en lista

```
import random as rand
lst = list(range(0, 10))
rand.choice(lst)
# Returnerar ett av talen i lst, taget på måfå. lst är oförändrad.
```