I Python är **operatorer** speciella symboler, kombinationer av symboler eller nyckelord som anger någon typ av beräkning. Du kan kombinera objekt och operatorer för att bygga **uttryck** som utför själva beräkningen. Så, operatorer är byggstenarna i uttryck, som du kan använda för att manipulera dina data. Därför är det viktigt för dig som programmerare att förstå hur operatörer fungerar i Python.

I den här handledningen kommer du att lära dig om de operatörer som Python för närvarande stöder. Du kommer också att lära dig grunderna i hur du använder dessa operatorer för att bygga uttryck.

I den här handledningen ska du:

- Lär känna Pythons **aritmetiska operatorer** och använd dem för att bygga **aritmetiska uttryck**
- Utforska Pythons **jämförelse-** , **booleska-** , **identitets-** och **medlemskapsoperatörer**
- Bygg **uttryck** med jämförelse-, booleska-, identitets- och medlemskapsoperatorer
- Lär dig mer om Pythons **bitvisa** operatorer och hur du använder dem
- Kombinera och upprepa sekvenser med hjälp av **sammanlänknings-** och **upprepningsoperatorerna**
- Förstå de **utökade uppdragsoperatörerna** och hur de fungerar

För att få ut det mesta av den här handledningen bör du ha en grundläggande förståelse för Python-programmeringskoncept, såsom variabler , tilldelningar och inbyggda datatyper .

Komma igång med operatorer och uttryck

I programmering är en **operatör** vanligtvis en symbol eller kombination av symboler som låter dig utföra en specifik operation. Denna operation kan verka på en eller flera **operander** . Om operationen involverar en enskild operand är operatören **unär** . Om operatorn involverar två operander är operatorn **binär** .

Till exempel, i Python kan du använda minustecknet (`-`) som en unär operator för att deklarera ett negativt tal. Du kan också använda den för att subtrahera två tal:

>>>

```
>>> -273.15
-273.15

>>> 5 - 2
3
```

I det här kodavsnittet är minustecknet (`-`) i det första exemplet en unär operator och numret `273.15` är operanden. I det andra exemplet är samma symbol en binär operator, och talen `5` och `2` är dess vänstra och högra operander.

Programmeringsspråk har vanligtvis operatörer inbyggda som en del av sin syntax. På många språk, inklusive Python, kan du också skapa din egen operatör eller ändra beteendet hos befintliga, vilket är en kraftfull och avancerad funktion att ha.

I praktiken tillhandahåller operatörer en snabb genväg för dig att manipulera data, utföra matematiska beräkningar, jämföra värden, köra booleska tester, tilldela värden till variabler och mer. I Python kan en operator vara en symbol, en kombination av symboler eller ett nyckelord beroende på vilken typ av operator du har att göra med.

Du har till exempel redan sett subtraktionsoperatorn, som representeras med ett enda minustecken ( ) `-` . Likhetsoperatorn är ett dubbelt likhetstecken (`==`). Så det är en kombination av symboler:

I det här exemplet använder du Python-likhetsoperatorn (`==`) för att jämföra två tal. Som ett resultat får du , `True` vilket är ett av Pythons booleska värden.

På tal om booleska värden, de booleska eller logiska operatorerna i Python är nyckelord snarare än tecken, som du kommer att lära dig i avsnittet om booleska operatorer och uttryck . Så istället för de udda tecknen som , , `||` och `&&` som `!` många andra programmeringsspråk använder, använder Python `or` , `and` , och `not` .

Att använda nyckelord istället för udda tecken är ett riktigt coolt designbeslut som överensstämmer med det faktum att Python älskar och uppmuntrar kodens läsbarhet .

Du hittar flera kategorier eller grupper av operatörer i Python. Här är en snabb lista över dessa kategorier:

- **Uppdragsoperatörer** _
- **Aritmetiska** operatorer
- **Jämförelseoperatörer** _
- **booleska** eller logiska operatorer
- **Identitetsoperatörer** _
- **Medlemskapsoperatörer** _
- **Sammankopplings-** och **repetitionsoperatorer** _
- **Bitvisa** operatörer

Alla dessa typer av operatörer tar hand om specifika typer av beräkningar och databearbetningsuppgifter. Du kommer att lära dig mer om dessa kategorier i denna handledning. Men innan du går in i mer praktiska diskussioner måste du veta att det mest elementära målet för en operatör är att vara en del av ett uttryck . Operatörer själva gör inte mycket:

>>>

```
>>> -
  File "<input>", line 1
    -
    ^
SyntaxError: incomplete input

>>> ==
  File "<input>", line 1
    ==
    ^^
SyntaxError: incomplete input

>>> or
  File "<input>", line 1
    or
    ^^
SyntaxError: incomplete input
```

Som du kan se i det här kodavsnittet, om du använder en operator utan de nödvändiga operanderna, får du ett syntaxfel . Så, operatorer måste vara en del av uttryck, som du kan bygga med Python-objekt som operander.

Så, vad är ett uttryck egentligen? Python har enkla och sammansatta uttalanden. En enkel sats är en konstruktion som upptar en enda logisk linje , som en tilldelningssats. En sammansatt sats är en konstruktion som upptar flera logiska linjer, till exempel en `for` loop eller en villkorlig sats. Ett **uttryck** är ett enkelt påstående som producerar och returnerar ett värde.

Du hittar operatorer i många uttryck. Här är några exempel:

>>>

```
>>> 7 + 5
12

>>> 42 / 2
21.0

>>> 5 == 5
True
```

I de två första exemplen använder du additions- och divisionsoperatorerna för att konstruera två aritmetiska uttryck vars operander är heltal. I det sista exemplet använder du likhetsoperatorn

för att skapa ett jämförelseuttryck. I alla fall får du ett specifikt värde efter exekvering av uttrycket.

Observera att inte alla uttryck använder operatorer. Till exempel är ett bara funktionsanrop ett uttryck som inte kräver någon operatör:

>>>

```
>>> abs(-7)
7

>>> pow(2, 8)
256

>>> print("Hello, World!")
Hello, World!
```

I det första exemplet anropar du den inbyggda `abs()` funktionen för att få det absoluta värdet av `-7` . Sedan beräknar du `2` kraften i `8` att använda den inbyggda `pow()` funktionen. Dessa funktionsanrop upptar en enda logisk rad och returnerar ett värde. Alltså, de är uttryck.

Slutligen är anropet till den inbyggda `print()` funktionen ett annat uttryck. Den här gången returnerar funktionen inte ett fruktbart värde, men den returnerar fortfarande , `None` vilket är Python- nulltypen . Så, samtalet är tekniskt sett ett uttryck.

Även om alla uttryck är påståenden, är inte alla påståenden uttryck. Till exempel returnerar rena uppdragssatser inget värde, vilket du kommer att lära dig på ett ögonblick. Därför är de inte uttryck. Tilldelningsoperatorn är en speciell operator som inte skapar ett uttryck utan ett uttalande.

Okej! Det var en snabb introduktion till operatorer och uttryck i Python. Nu är det dags att dyka djupare in i ämnet. För att kicka igång börjar du med uppdragsoperatören och uttalanden.

Uppdragsoperatören och uttalanden

Tilldelningsoperatorn är en av de mest använda operatorerna i Python **.** Operatören består av ett enda likhetstecken (`=`), och den fungerar på två operander. Vänsteroperanden är vanligtvis en variabel medan den högra operanden är ett uttryck.

Tilldelningsoperatorn låter dig tilldela *värden till variabler* . Strängt taget, i Python, gör denna operator att variabler eller namn refererar till specifika objekt i din dators minne. Med andra ord, en uppgift skapar en referens till ett konkret objekt och kopplar den referensen till målvariabeln.

Till exempel skapar alla påståenden nedan nya variabler som innehåller referenser till specifika objekt:

>>>

```
>>> number = 42
>>> day = "Friday"
>>> digits = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
>>> letters = ["a", "b", "c"]
```

I den första satsen skapar du `number` variabeln, som innehåller en referens till numret `42` i din dators minne. Man kan också säga att namnet `number` pekar på `42` , som är ett konkret föremål.

I resten av exemplen skapar du andra variabler som pekar på andra typer av objekt, som en sträng , tupel respektive lista .

Du kommer att använda tilldelningsoperatorn i många av exemplen som du kommer att skriva i den här handledningen. Ännu viktigare, du kommer att använda den här operatorn många gånger i din egen kod. Det kommer att vara din eviga vän. Nu kan du dyka in i andra Python-operatorer!

Aritmetiska operatorer och uttryck i Python

Aritmetiska operatorer är de operatorer som låter dig utföra *aritmetiska operationer* på numeriska värden. Ja, de kommer från matematik, och i de flesta fall representerar du dem med de vanliga matematiska tecknen. Följande tabell listar de aritmetiska operatorer som Python för närvarande stöder:

Operatör	Typ	Drift	Exempel på uttryck	Resultat
`+`	Unary	Positiv	`+a`	`a` utan någon transformation eftersom detta helt enkelt är ett komplement till negation
`+`	Binär	Tillägg	`a + b`	Den aritmetiska summan av `a` och `b`
`-`	Unary	Negation	`-a`	Värdet av `a` men med motsatt tecken
`-`	Binär	Subtraktion	`a - b`	`b` subtraherad från `a`
`*`	Binär	Multiplikation	`a * b`	Produkten av `a` och `b`
`/`	Binär	Division	`a / b`	Kvoten av `a` dividerat med `b`, uttryckt som en float
`%`	Binär	Modulo	`a % b`	Resten av `a` dividerat med `b`
`//`	Binär	Golvdelning eller heltalsdelning	`a // b`	Kvoten av `a` dividerat med `b`, avrundat till näst minsta heltal
`**`	Binär	Exponentiering	`a**b`	`a` upphöjd till makten av `b`

Observera att `a` och `b` i kolumnen *Exempeluttryck* representerar numeriska värden, som heltal , flyttal , komplexa , rationella och decimala tal.

Här är några exempel på dessa operatörer som används:

>>>

```
>>> a = 5
>>> b = 2

>>> +a
5
>>> -b
-2
>>> a + b
7
>>> a - b
3
>>> a * b
10
>>> a / b
2.5
>>> a % b
1
>>> a // b
2
>>> a**b
25
```

I det här kodavsnittet skapar du först två nya variabler, `a` och `b`, håller `5` och `2`, respektive. Sedan använder du dessa variabler för att skapa olika aritmetiska uttryck med en specifik operator i varje uttryck.

Återigen, standarddivisionsoperatören (`/`) returnerar alltid ett flyttal, även om utdelningen är jämnt delbar med divisorn:

>>>

```
>>> 10 / 5
2.0

>>> 10.0 / 5
2.0
```

I det första exemplet `10` är jämnt delbart med `5`. Därför kan denna operation returnera heltal `2`. Den returnerar dock flyttalstalet `2.0`. I det andra exemplet `10.0` är ett flyttalstal och `5` är ett heltal. I det här fallet främjar Python internt `5` till `5.0` och driver divisionen. Resultatet är också ett flyttal.

`//` Tänk slutligen på följande exempel på hur du använder operatorn våningsindelning ( ):

>>>

```
>>> 10 // 4
2
>>> -10 // -4
2

>>> 10 // -4
-3
>>> -10 // 4
-3
```

Golvindelning avrundar alltid nedåt . Det betyder att resultatet är det största heltal som är mindre än eller lika med kvoten. För positiva tal är det som om bråkdelen är trunkerad och bara heltalsdelen kvarstår.

Jämförelseoperatorer och uttryck i Python

Python- **jämförelseoperatorerna** låter dig *jämföra* numeriska värden och andra objekt som stöder dem. Tabellen nedan listar alla för närvarande tillgängliga jämförelseoperatorer i Python:

Operatör	Drift	Exempel på uttryck	Resultat
`==`	Lika med	`a == b`	• `True` om värdet på `a` är lika med värdet på `b`
• `False` annars			
`!=`	Inte lika med	`a != b`	• `True` om `a` inte är lika med `b`
• `False` annars			
`<`	Mindre än	`a < b`	• `True` om `a` är mindre än `b`
• `False` annars			
`<=`	Mindre än eller lika med	`a <= b`	• `True` om `a` är mindre än eller lika med `b`
• `False` annars			
`>`	Större än	`a > b`	• `True` om `a` är större än `b`
• `False` annars			
`>=`	Större än eller lika med	`a >= b`	• `True` om `a` är större än eller lika med `b`
• `False` annars			

Jämförelseoperatorerna är alla binära. Detta innebär att de kräver vänster och höger operander. Dessa operatorer returnerar alltid ett booleskt värde (`True` eller `False`) som beror på sanningsvärdet för den aktuella jämförelsen.

Observera att jämförelser mellan objekt av olika datatyper ofta inte är meningsfulla och ibland inte är tillåtna i Python. Du kan till exempel jämföra ett nummer och en sträng för likhet med operatorn `==` . Däremot får du `False` som ett resultat:

Heltalet `2` är inte lika med strängen `"2"` . Därför får du `False` som ett resultat. Du kan också använda `!=` operatorn i uttrycket ovan, i vilket fall du får `True` som ett resultat.

Icke-likvärdiga jämförelser mellan operander av olika datatyper ger ett `TypeError` undantag:

```
>>>
```

```
>>> 5 < "7"
Traceback (most recent call last):
    ...
TypeError: '<' not supported between instances of 'int' and 'str'
```

I det här exemplet höjer Python ett `TypeError` undantag eftersom en mindre än jämförelse (`<`) inte är vettigt mellan ett heltal och en sträng. Så operationen är inte tillåten.

Det är viktigt att notera att i samband med jämförelser är heltals- och flyttalsvärden kompatibla, och du kan jämföra dem.

Du kommer vanligtvis att använda och hitta jämförelseoperatorer i booleska sammanhang som villkorliga uttalanden och `while` loopar . De låter dig fatta beslut och definiera ett programs kontrollflöde .

Jämförelseoperatorerna fungerar på flera typer av operander, såsom siffror, strängar, tupler och listor. I följande avsnitt kommer du att utforska skillnaderna.

Jämförelse av heltalsvärden

Förmodligen är de mer enkla jämförelserna i Python och i matematik de som involverar heltal. De låter dig räkna riktiga föremål, vilket är en välbekant daglig uppgift. Faktum är att de icke-negativa heltalen också kallas naturliga tal . Så att jämföra den här typen av nummer är förmodligen ganska intuitivt, och att göra det i Python är inget undantag.

Betrakta följande exempel som jämför heltal:

>>>

```
>>> a = 10
>>> b = 20
>>> a == b
False
>>> a != b
True
>>> a < b
True
>>> a <= b
True
>>> a > b
False
>>> a >= b
False

>>> x = 30
>>> y = 30
>>> x == y
True
>>> x != y
False
>>> x < y
False
>>> x <= y
True
>>> x > y
False
>>> x >= y
True
```

I den första uppsättningen exempel definierar du två variabler `a` och `b`, för att göra några jämförelser mellan dem. Värdet på `a` är mindre än värdet på `b`. Så varje jämförelseuttryck returnerar det förväntade booleska värdet. Den andra uppsättningen exempel använder två värden som är lika, och återigen får du de förväntade resultaten.

Jämförelse av flyttalsvärden

Att jämföra flyttalstal är lite mer komplicerat än att jämföra heltal. Värdet som lagras i ett `float` objekt kanske inte är exakt vad du tror att det skulle vara. Av den anledningen är det dålig praxis att jämföra flyttalsvärden för exakt jämlikhet med hjälp av `==` operatorn.

Tänk på exemplet nedan:

>>>

```
>>> x = 1.1 + 2.2
>>> x == 3.3
False

>>> 1.1 + 2.2
3.3000000000000003
```

Hoppsan! Den interna representationen av detta tillägg är inte exakt lika med , `3.3` som du kan se i det sista exemplet. Så jämför `x` med `3.3` jämställdhetsoperatören avkastning `False` .

För att jämföra flyttalstal för jämlikhet måste du använda ett annat tillvägagångssätt. Det föredragna sättet att avgöra om två flyttalsvärden är lika är att avgöra om de ligger nära varandra, givet viss tolerans.

Modulen `math` från standardbiblioteket tillhandahåller en funktion som bekvämt kallas `isclose()` som hjälper dig med `float` jämförelser. Funktionen tar två tal och testar dem för ungefärlig likhet:

>>>

```
>>> from math import isclose

>>> x = 1.1 + 2.2

>>> isclose(x, 3.3)
True
```

I det här exemplet använder du `isclose()` funktionen för att jämföra `x` och `3.3` för ungefärlig likhet. Den här gången får du `True` som ett resultat eftersom båda siffrorna är tillräckligt nära för att anses lika.

För ytterligare information om hur du använder `isclose()` , kolla in sektionen Hitta närhet till tal med Python `isclose()` i Python- `math` modulen: Allt du behöver veta .

Jämförelse av strängar

Du kan också använda jämförelseoperatorerna för att jämföra Python-strängar i din kod. I detta sammanhang måste du vara medveten om hur Python internt jämför strängobjekt. I praktiken jämför Python strängar tecken för tecken med varje teckens Unicode- **kodpunkt** . Unicode är Pythons standardteckenuppsättning .

Du kan använda den inbyggda `ord()` funktionen för att lära dig Unicode-kodpunkten för vilket tecken som helst i Python. Tänk på följande exempel:

>>>

```
>>> ord("A")
65
>>> ord("a")
97

>>> "A" == "a"
False
>>> "A" > "a"
False
>>> "A" < "a"
True
```

Versalerna `"A"` har en lägre Unicode-punkt än gemenerna `"a"`. Alltså `"A"` är mindre än `"a"`. Till slut jämför Python tecken med heltal. Så samma regler som Python använder för att jämföra heltal gäller för strängjämförelse.

När det kommer till strängar med flera tecken kör Python jämförelsen tecken för tecken i en loop.

Jämförelsen använder lexikografisk ordning , vilket innebär att Python jämför det första objektet från varje sträng. Om deras Unicode-kodpunkter är olika, avgör denna skillnad jämförelseresultatet. Om Unicode-kodpunkterna är lika, jämför Python de kommande två tecknen, och så vidare, tills endera strängen är slut:

>>>

```
>>> "Hello" > "HellO"
True

>>> ord("o")
111
>>> ord("O")
79
```

I det här exemplet jämför Python båda operanderna tecken för tecken. När den når slutet av strängen jämför den `"o"` och `"O"`. Eftersom den gemena bokstaven har en större Unicode-kodpunkt är den första versionen av strängen större än den andra.

Du kan också jämföra strängar av olika längder:

>>>

```
>>> "Hello" > "Hello, World!"
False
```

I det här exemplet kör Python en jämförelse tecken för tecken som vanligt. Om det tar slut på tecken, är den kortare strängen mindre än den längre. Detta betyder också att den tomma strängen är den minsta möjliga strängen.

Jämförelse av listor och tupler

I din Python-resa kan du också möta behovet av att jämföra listor med andra listor och tupler med andra tupler. Dessa datatyper stöder också standardjämförelseoperatorerna. Precis som med strängar, när du använder en jämförelseoperator för att jämföra två listor eller två tupler, kör Python en jämförelse objekt för objekt.

Observera att Python tillämpar specifika regler beroende på typen av objekt som ingår. Här är några exempel som jämför listor och tuplar av heltalsvärden:

>>>

```
>>> [2, 3] == [2, 3]
True
>>> (2, 3) == (2, 3)
True

>>> [5, 6, 7] < [7, 5, 6]
True
>>> (5, 6, 7) < (7, 5, 6)
True

>>> [4, 3, 2] < [4, 3, 2]
False
>>> (4, 3, 2) < (4, 3, 2)
False
```

I dessa exempel jämför du listor och tuplar med siffror med hjälp av standardjämförelseoperatorerna. När du jämför dessa datatyper kör Python en jämförelse för objekt.

Till exempel, i det första uttrycket ovan, jämför Python den `2` vänstra operanden och den `2` högra operanden. Eftersom de är lika, fortsätter Python att jämföra `3` och `3` dra slutsatsen att båda listorna är lika. Samma sak händer i det andra exemplet, där du jämför tupler som innehåller samma data.

Det är viktigt att notera att du faktiskt kan jämföra listor med tupler med hjälp av operatorerna `==` och . `!=` Du kan dock *inte* jämföra listor och tupler med operatorerna `<` , `>` , `<=` , och : `>=`

>>>

```
>>> [2, 3] == (2, 3)
False
>>> [2, 3] != (2, 3)
True

>>> [2, 3] > (2, 3)
Traceback (most recent call last):
    ...
TypeError: '>' not supported between instances of 'list' and 'tuple'

>>> [2, 3] <= (2, 3)
Traceback (most recent call last):
    ...
TypeError: '<=' not supported between instances of 'list' and 'tuple'
```

Python stöder jämställdhetsjämförelse mellan listor och tupler. Det stöder dock inte resten av jämförelseoperatorerna, som du kan dra slutsatsen från de två sista exemplen. Om du försöker använda dem får du ett `TypeError` meddelande om att operationen inte stöds.

Du kan också jämföra listor och tupler av olika längder:

>>>

```
>>> [5, 6, 7] < [8]
True
>>> (5, 6, 7) < (8,)
True

>>> [5, 6, 7] == [5]
False
>>> (5, 6, 7) == (5,)
False

>>> [5, 6, 7] > [5]
True
>>> (5, 6, 7) > (5,)
True
```

I de två första exemplen får du `True` som ett resultat eftersom `5` är mindre än `8`. Det faktum är tillräckligt för att Python ska lösa jämförelsen. I det andra paret av exempel får du `False`. Detta resultat är vettigt eftersom de jämförda sekvenserna inte har samma längd, så de kan inte vara lika.

I det sista exemplet jämför Python `5` med `5`. De är lika, så jämförelsen fortsätter. Eftersom det inte finns fler värden att jämföra i högeroperanderna drar Python slutsatsen att vänsterhandsoperanderna är större.

Som du kan se kan det vara svårt att jämföra listor och tupler. Det är också en dyr operation som i värsta fall kräver att man korsar två hela sekvenser. Saker och ting blir mer komplexa och dyrare när de inneslutna föremålen också är sekvenser. I dessa situationer måste Python också jämföra objekt på ett värde-för-värde sätt, vilket ökar kostnaden för operationen.

Booleska operatorer och uttryck i Python

Python har tre booleska eller logiska operatorer: , `and`, `or` och `not`. De definierar en uppsättning operationer som betecknas av de generiska operatörerna `AND`, `OR` och `NOT`. Med dessa operatorer kan du skapa sammansatta villkor.

I följande avsnitt kommer du att lära dig hur Python Boolean-operatorerna fungerar. Speciellt kommer du att lära dig att vissa av dem beter sig annorlunda när du använder dem med booleska värden eller med vanliga objekt som operander.

Booleska uttryck som involverar booleska operander

Du hittar många objekt och uttryck som är av boolesk typ eller , `bool` som Python kallar denna typ. Med andra ord utvärderar många objekt till `True` eller `False`, som är Python Boolean-värden.

Till exempel, när du utvärderar ett uttryck med en jämförelseoperator, är resultatet av det uttrycket alltid av `bool` typen:

>>>

```
>>> age = 20

>>> is_adult = age > 18
>>> is_adult
True

>>> type(is_adult)
<class 'bool'>
```

I det här exemplet returnerar uttrycket `age > 18` ett booleskt värde, som du lagrar i `is_adult` variabeln. Nu `is_adult` är av `bool` typ, som du kan se efter att ha anropat den inbyggda `type()` funktionen.

Du kan också hitta Python inbyggda och anpassade funktioner som returnerar ett booleskt värde. Denna typ av funktion är känd som en predikatfunktion . De inbyggda `all()` , `any()` , `callable()` , och `isinstance()` funktionerna är alla bra exempel på denna praxis.

Tänk på följande exempel:

>>>

```
>>> number = 42

>>> validation_conditions = (
...     isinstance(number, int),
...     number % 2 == 0,
... )

>>> all(validation_conditions)
True

>>> callable(number)
False
>>> callable(print)
True
```

I det här kodavsnittet definierar du först en variabel som kallas `number` med hjälp av din gamla vän tilldelningsoperatorn. Sedan skapar du en annan variabel som heter `validation_conditions` . Denna variabel har en tupel av uttryck. Det första uttrycket används `isinstance()` för att kontrollera om det `number` är ett heltalsvärde.

Det andra är ett sammansatt uttryck som kombinerar operatorerna modulo (`%`) och likhet (`==`) för att skapa ett villkor som kontrollerar om inmatningsvärdet är ett jämnt tal. I detta tillstånd returnerar modulo-operatorn resten av att dividera `number` med `2` , och likhetsoperatorn jämför resultatet med , `0` returnerar `True` eller `False` som jämförelsens resultat.

Sedan använder du `all()` funktionen för att avgöra om alla villkor är sanna. I det här exemplet, eftersom `number = 42` , är villkoren sanna och `all()` returnerar `True` . Du kan leka med värdet av `number` om du vill experimentera lite.

I de två sista exemplen använder du `callable()` funktionen. Som namnet antyder låter den här funktionen dig avgöra om ett objekt är **anropbart** . Att vara anropsbar betyder att du kan

anropa objektet med ett par parenteser och lämpliga argument, som du skulle kalla vilken Python-funktion som helst.

Variabeln `number` är inte anropsbar, och funktionen returnerar `False` därför. Däremot `print()` är funktionen anropbar, så `callable()` returnerar `True`.

All tidigare diskussion är grunden för att förstå hur Python logiska operatorer fungerar med booleska operander.

Logiska uttryck som involverar , `and` , `or` och `not` är enkla när operanderna är booleska. Här är en sammanfattning. Observera att `x` och `y` representerar booleska operander:

Operatör	**Exempel på uttryck**	**Resultat**
`and`	`x and y`	• `True` om både `x` och `y` är `True`
• `False` annars		
`or`	`x or y`	• `True` om antingen `x` eller `y` är `True`
• `False` annat		
`not`	`not x`	• `True` om `x` är `False`
• `False` om `x` är `True`		

Den här tabellen sammanfattar sanningsvärdet för uttryck som du kan skapa med de logiska operatorerna med booleska operander. Det finns något att notera i denna sammanfattning. Till skillnad från `and` och `or` , som är binära operatorer, `not` är operatorn unär, vilket betyder att den fungerar på en operand. Denna operand måste alltid vara på höger sida.

Nu är det dags att ta en titt på hur operatörerna arbetar i praktiken. Här är några exempel på hur `and` operatorn används med booleska operander:

>>>

```
>>> 5 < 7 and 3 == 3
True

>>> 5 < 7 and 3 != 3
False

>>> 5 > 7 and 3 == 3
False

>>> 5 > 7 and 3 != 3
False
```

I det första exemplet returnerar båda operanderna `True` . Därför `and` återkommer uttrycket `True` som ett resultat. I det andra exemplet är den vänstra operanden , `True` men den högra

operanden är `False`. På grund av detta `and` återkommer operatören `False`.

I det tredje exemplet är den vänstra operanden `False`. I detta fall `and` återkommer operatören omedelbart `False` och utvärderar aldrig `3 == 3` tillståndet. Detta beteende kallas kortslutningsutvärdering. Du kommer att lära dig mer om det på ett ögonblick.

I det sista exemplet returnerar båda villkoren `False`. Återigen, `and` återvänder `False` som ett resultat. Men på grund av kortslutningsutvärderingen utvärderas inte det högra uttrycket.

Hur är det med `or` operatören? Här är några exempel som visar hur det fungerar:

>>>

```
>>> 5 < 7 or 3 == 3
True

>>> 5 < 7 or 3 != 3
True

>>> 5 > 7 or 3 == 3
True

>>> 5 > 7 or 3 != 3
False
```

I de tre första exemplen returnerar minst ett av villkoren `True`. I alla fall `or` återkommer operatören `True`. Observera att om den vänstra operanden är `True`, tillämpar `or` kortslutningsutvärderingen och utvärderar inte den högra operanden. Detta är vettigt. Om den vänstra operanden är , vet `True` du `or` redan slutresultatet. Varför skulle det behöva fortsätta utvärderingen om resultatet inte ändras?

I det sista exemplet är båda operanderna , `False` och detta är den enda situationen där `or` returnerar `False`. Det är viktigt att notera att om den vänstra operanden är , `False` då `or` måste man utvärdera den högra operanden för att komma fram till en slutlig slutsats.

Slutligen har du `not` operatorn, som förnekar det aktuella sanningsvärdet för ett objekt eller uttryck:

>>>

```
>>> 5 < 7
True

>>> not 5 < 7
False
```

Om du placerar `not` före ett uttryck får du det omvända sanningsvärdet. När uttrycket kommer tillbaka `True` får du `False`. När uttrycket utvärderas till `False` får du `True`.

Det finns en grundläggande beteendeskillnad mellan `not` och de andra två booleska operatorerna. I ett `not` uttryck får du alltid ett booleskt värde som resultat. Det är inte alltid regeln som styr `and` och `or` uttryck, som du kommer att lära dig i avsnittet Booleska uttryck som involverar andra typer av operander .

Utvärdering av vanliga objekt i ett booleskt sammanhang

I praktiken är de flesta Python-objekt och uttryck inte booleska. Med andra ord, de flesta objekt och uttryck har inte ett `True` eller `False` värde utan en annan typ av värde. Du kan dock använda vilket Python-objekt som helst i en boolesk kontext, till exempel en villkorssats eller en `while` loop.

I Python har alla objekt ett specifikt sanningsvärde. Så du kan använda de logiska operatorerna med alla typer av operander.

Python har väletablerade regler för att bestämma sanningsvärdet för ett objekt när du använder det objektet i en boolesk kontext eller som en operand i ett uttryck byggt med logiska operatorer. Så här står det i dokumentationen om detta ämne:

> Som standard anses ett objekt vara sant om inte dess klass definierar antingen en `__bool__()` metod som returnerar `False` eller en `__len__()` metod som returnerar noll, när den anropas med objektet. Här är de flesta av de inbyggda objekten som anses vara falska:
>
> - konstanter definierade som falska: `None` och `False`.
> - noll av valfri numerisk typ: `0`, `0.0`, `0j`, `Decimal(0)`, `Fraction(0, 1)`
> - tomma sekvenser och samlingar: `''`, `()`, `[]`, `{}`, `set()`, `range(0)`
>
> (Källa)

Du kan bestämma sanningsvärdet för ett objekt genom att anropa den inbyggda `bool()` funktionen med det objektet som argument. Om `bool()` returnerar `True` är objektet **sanning**. Om `bool()` returer `False` är det **falskt**.

För numeriska värden har du att ett nollvärde är falskt, medan ett värde som inte är noll är sant:

>>>

```
>>> bool(0), bool(0.0), bool(0.0+0j)
(False, False, False)

>>> bool(-3), bool(3.14159), bool(1.0+1j)
(True, True, True)
```

Python anser att nollvärdet för alla numeriska typer är falskt. Alla andra värden är sanna, oavsett hur nära noll de är.

När det gäller att utvärdera strängar har du att en tom sträng alltid är falsk, medan en icke-tom sträng är sann:

>>>

```
>>> bool("")
False

>>> bool(" ")
True

>>> bool("Hello")
True
```

Observera att strängar som innehåller vita blanksteg också är sanna i Pythons ögon. Så, blanda inte ihop tomma strängar med blankstegssträngar.

Slutligen är inbyggda behållardatatyper, såsom listor, tupler, uppsättningar och ordböcker , falska när de är tomma. Annars anser Python dem som sanna objekt:

>>>

```
>>> bool([])
False
>>> bool([1, 2, 3])
True

>>> bool(())
False
>>> bool(("John", 25, "Python Dev"))
True

>>> bool(set())
False
>>> bool({"square", "circle", "triangle"})
True

>>> bool({})
False
>>> bool({"name": "John", "age": 25, "job": "Python Dev"})
True
```

För att bestämma sanningsvärdet för containerdatatyper förlitar sig Python på den `.__len__()` speciella metoden . Denna metod ger stöd för den inbyggda `len()` funktionen, som du kan använda för att bestämma antalet föremål i en given behållare.

I allmänhet, om `.__len__()` returnerar `0`, betraktar Python behållaren som ett falskt objekt, vilket överensstämmer med de allmänna reglerna du just har lärt dig tidigare.

All diskussion om sanningsvärdet av Python-objekt i det här avsnittet är nyckeln till att förstå hur de logiska operatorerna beter sig när de tar godtyckliga objekt som operander.

Booleska uttryck som involverar andra typer av operander

Du kan också använda alla objekt, som siffror eller strängar, som operander till , `and` , `or` och `not` . Du kan till och med använda kombinationer av ett booleskt objekt och ett vanligt. I dessa situationer beror resultatet på operandernas sanningsvärde.

Du har redan lärt dig hur Python bestämmer sanningsvärdet för objekt. Så du är redo att dyka in i att skapa uttryck med logiska operatorer och vanliga objekt.

Till att börja med, nedan är en tabell som sammanfattar vad du får när du använder två objekt, `x` och `y` i ett `and` uttryck:

Om `x` är	`x and y` returnerar
Sanning	`y`
Falskt	`x`

Det är viktigt att betona en subtil detalj i tabellen ovan. När du använder `and` i ett uttryck får du inte alltid `True` eller `False` som ett resultat. Istället får du en av operanderna. Du får bara `True` eller `False` om den returnerade operanden har något av dessa värden.

Här är några kodexempel som använder heltalsvärden. Kom ihåg att i Python är nollvärdet för numeriska typer falskt. Resten av värdena är sanna:

>>>

```
>>> 3 and 4
4

>>> 0 and 4
0

>>> 3 and 0
0
```

I det första uttrycket är den vänstra operanden (`3`) sann. Så du får den högra operanden (`4`) som ett resultat.

I det andra exemplet är den vänstra operanden (`0`) falsk, och du får den som ett resultat. I det här fallet tillämpar Python kortslutningsutvärderingstekniken. Den vet redan att hela uttrycket är falskt eftersom `0` det är falskt, så Python returnerar `0` omedelbart utan att utvärdera den högra operanden.

I det slutliga uttrycket är den vänstra operanden (`3`) sann. Därför behöver Python utvärdera den högra operanden för att dra en slutsats. Som ett resultat får du den högra operanden, oavsett vad dess sanningsvärde är.

När det kommer till att använda `or` operatorn får du också en av operanderna som resultat. Detta är vad som händer för två godtyckliga objekt, `x` och `y` :

Om `x` är	`x or y` returnerar
Sanning	`x`
Falskt	`y`

Återigen, uttrycket `x or y` evalueras inte till antingen `True` eller `False` . Istället returnerar den en av dess operander, `x` eller `y` .

Som du kan dra slutsatsen från ovanstående tabell, om den vänstra operanden är sann, får du den som ett resultat. Annars får du den andra operanden. Här är några exempel som visar detta beteende:

>>>

```
>>> 3 or 4
3

>>> 0 or 4
4

>>> 3 or 0
3
```

I det första exemplet är den vänstra operanden sann, och `or` returnerar den omedelbart. I det här fallet utvärderar Python inte den andra operanden eftersom den redan känner till slutresultatet. I det andra exemplet är den vänstra operanden falsk, och Python måste utvärdera den högra för att bestämma resultatet.

I det sista exemplet är den vänstra operanden sann, och det faktumet definierar resultatet av uttrycket. Det finns inget behov av att utvärdera den högra operanden.

Ett uttryck som `x or y` är sant om antingen `x` eller `y` är sant, och falskt om både `x` och `y` är falska. Denna typ av uttryck returnerar den första sanningsoperand som den hittar. Om båda operanderna är falska, returnerar uttrycket den högra operanden. För att se det senare beteendet i praktiken, överväg följande exempel:

I detta specifika uttryck är båda operanderna falska. Så, `or` operatorn returnerar den högra operanden, och hela uttrycket är falskt som ett resultat.

Slutligen har du `not` operatören. Du kan också använda den här med vilket objekt som helst som en operand. Så här händer:

Om `x` är	**`not x` returnerar**
Sanning	`False`
Falskt	`True`

Operatören `not` har ett enhetligt beteende. Det returnerar alltid ett booleskt värde. Detta beteende skiljer sig från sina syskonoperatörer `and` och `or`.

Här är några kodexempel:

>>>

```
>>> not 3
False

>>> not 0
True
```

I det första exemplet är operanden, , `3` sann från Pythons synvinkel. Så operatören återvänder `False`. I det andra exemplet är operanden falsk och `not` returnerar `True`.

Sammanfattningsvis `not` negerar Python-operatorn sanningsvärdet för ett objekt och returnerar alltid ett booleskt värde. Det senare beteendet skiljer sig från beteendet hos syskonoperatörerna `and` och `or`, som returnerar operander snarare än booleska värden.

Sammansatta logiska uttryck och kortslutningsutvärdering

Hittills har du sett uttryck med bara en singel `or` eller `and` operator och två operander. Men du kan också skapa sammansatta logiska uttryck med flera logiska operatorer och operander.

För att illustrera hur man skapar ett sammansatt uttryck med , `or` överväg följande leksaksexempel:

```
x1 or x2 or x3 or ... or xn
```

Detta uttryck returnerar det första sanningsvärdet. Om alla föregående `x` variabler är falska, returnerar uttrycket det sista värdet, `xn`.

För att visa kortslutningsutvärdering, anta att du har en identitetsfunktion , , `f()` som beter sig enligt följande:

- Tar ett enda argument
- Visar funktionen och dess argument på skärmen
- Returnerar argumentet som dess returvärde

Här är koden för att definiera den här funktionen och även några exempel på hur den fungerar:

>>>

```
>>> def f(arg):
...     print(f"-> f({arg}) = {arg}")
...     return arg
...

>>> f(0)
-> f(0) = 0
0

>>> f(False)
-> f(False) = False
False

>>> f(1.5)
-> f(1.5) = 1.5
1.5
```

Funktionen `f()` visar sitt argument, som visuellt bekräftar om du anropade funktionen. Det returnerar också argumentet när du skickade det i samtalet. På grund av detta beteende kan du få uttrycket `f(arg)` att vara sant eller falskt genom att ange ett värde för `arg` det som är sant eller falskt.

Tänk nu på följande sammansatta logiska uttryck:

>>>

```
>>> f(0) or f(False) or f(1) or f(2) or f(3)
-> f(0) = 0
-> f(False) = False
-> f(1) = 1
1
```

I det här exemplet utvärderar Python först , `f(0)` vilket returnerar `0` . Detta värde är falskt. Uttrycket är inte sant än, så utvärderingen fortsätter från vänster till höger. Nästa operand, `f(False)` , returnerar `False` . Det värdet är också falskt, så utvärderingen fortsätter.

Nästa upp är `f(1)` . Det utvärderar till `1` , vilket är sant. Vid den tidpunkten stoppar Python utvärderingen eftersom den redan vet att hela uttrycket är sant. Följaktligen returnerar Python `1` som uttryckets värde och utvärderar aldrig de återstående operanderna, `f(2)` och `f(3)` . Du kan bekräfta från utgången att anropen `f(2)` och `f(3)` inte förekommer.

Ett liknande beteende visas i ett uttryck med flera `and` operatorer som följande:

```
x1 and x2 and x3 and ... and xn
```

Detta uttryck är sant om alla operander är sanna. Om minst en operand är falsk, är uttrycket också falskt.

I det här exemplet dikterar kortslutningsutvärdering att Python slutar utvärdera så snart en operand råkar vara falsk. Vid den tidpunkten är hela uttrycket känt för att vara falskt. När så är fallet, slutar Python att utvärdera operander och returnerar den falska operanden som avslutade utvärderingen.

Här är två exempel som bekräftar kortslutningsbeteendet:

>>>

```
>>> f(1) and f(False) and f(2) and f(3)
-> f(1) = 1
-> f(False) = False
False

>>> f(1) and f(0.0) and f(2) and f(3)
-> f(1) = 1
-> f(0.0) = 0.0
0.0
```

I båda exemplen stannar utvärderingen vid den första falska termen – `f(False)` i det första fallet, `f(0.0)` i det andra fallet – och varken anropet `f(2)` eller anropet `f(3)` inträffar. I slutändan återkommer uttrycken `False` och `0.0` resp.

Om alla operander är sanna, utvärderar Python dem alla och returnerar den sista (längst till höger) som värdet på uttrycket:

>>>

```
>>> f(1) and f(2.2) and f("Hello")
-> f(1) = 1
-> f(2.2) = 2.2
-> f(Hello) = Hello
'Hello'

>>> f(1) and f(2.2) and f(0)
-> f(1) = 1
-> f(2.2) = 2.2
-> f(0) = 0
0
```

I det första exemplet är alla operander sanna. Uttrycket är också sant och ger den sista operanden. I det andra exemplet är alla operander sanna utom den sista. Uttrycket är falskt och returnerar den sista operanden.

Idiom som utnyttjar kortslutningsutvärdering

När du gräver i Python kommer du att upptäcka att det finns några vanliga idiomatiska mönster som utnyttjar kortslutningsutvärdering för att uttrycka, prestanda och säkerheten kortfattat. Du kan till exempel dra nytta av den här typen av utvärdering för:

- Undviker ett undantag
- Anger ett standardvärde
- Att hoppa över en kostsam operation

För att illustrera den första punkten, anta att du har två variabler, `a` och `b`, och du vill veta om divisionen av `b` med `a` resulterar i ett tal större än `0`. I det här fallet kan du köra följande uttryck eller villkor:

>>>

```
>>> a = 3
>>> b = 1

>>> (b / a) > 0
True
```

Den här koden fungerar. Du måste dock ta hänsyn till möjligheten som `a` kan vara `0`, i så fall får du ett undantag :

>>>

```
>>> a = 0
>>> b = 1

>>> (b / a) > 0
Traceback (most recent call last):
    ...
ZeroDivisionError: division by zero
```

I det här exemplet är divisorn , `0` vilket gör att Python höjer ett `ZeroDivisionError` undantag. Detta undantag bryter din kod. Du kan hoppa över det här felet med ett uttryck som följande:

>>>

```
>>> a = 0
>>> b = 1

>>> a != 0 and (b / a) > 0
False
```

När `a` är `0` , `a != 0` är falskt. Pythons kortslutningsutvärdering säkerställer att utvärderingen stannar vid den punkten, vilket innebär att den `(b / a)` aldrig körs och att felet aldrig uppstår.

Med den här tekniken kan du implementera en funktion för att avgöra om ett heltal är delbart med ett annat heltal:

```
def is_divisible(a, b):
    return b != 0 and a % b == 0
```

I den här funktionen, om `b` är `0` , `a / b` är inte definierad. Så siffrorna är inte delbara. Om `b` skiljer sig från `0` , kommer resultatet att bero på resten av divisionen.

Att välja ett standardvärde när ett angivet värde är falskt är ett annat idiom som drar fördel av kortslutningsutvärderingsfunktionen hos Pythons logiska operatorer.

Säg till exempel att du har en variabel som ska innehålla ett lands namn. Vid någon tidpunkt kan denna variabel sluta hålla en tom sträng. Om så är fallet vill du att variabeln ska ha ett standard länsnamn. Du kan också göra detta med `or` operatören:

>>>

```
>>> country = "Canada"
>>> default_country = "United States"

>>> country or default_country
'Canada'

>>> country = ""
>>> country or default_country
'United States'
```

Om `country` den inte är tom, så är den sann. I det här scenariot kommer uttrycket att returnera det första sanningsvärdet, som finns `country` i det första `or` uttrycket. Utvärderingen stannar, och du får `"Canada"` som resultat.

Å andra sidan, om `country` är en tom sträng , så är det falskt. Utvärderingen fortsätter till nästa operand, , `default_country` vilket är sant. Slutligen får du standardlandet som ett resultat.

Ett annat intressant användningsfall för kortslutningsutvärdering är att undvika kostsamma operationer samtidigt som man skapar sammansatta logiska uttryck. Till exempel, om du har en kostsam operation som bara ska köras om ett givet villkor är falskt, kan du använda `or` som i följande kodavsnitt:

```
data_is_clean or clean_data(data)
```

I denna konstruktion `clean_data()` representerar din funktion en kostsam operation. På grund av kortslutningsutvärdering kommer den här funktionen bara att köras när den `data_is_clean` är falsk, vilket betyder att din data inte är ren.

En annan variant av denna teknik är när du vill köra en kostsam operation om ett givet villkor är sant. I det här fallet kan du använda `and` operatorn:

```
data_is_updated and process_data(data)
```

I det här exemplet `and` utvärderar operatören `data_is_updated` . Om denna variabel är sann fortsätter utvärderingen och `process_data()` funktionen körs. Annars stannar utvärderingen och `process_data()` körs aldrig.

Sammansatta vs kedjade uttryck

Ibland har du ett sammansatt uttryck som använder `and` operatorn för att sammanfoga jämförelseuttryck. Säg till exempel att du vill avgöra om ett tal är i ett givet intervall. Du kan lösa det här problemet med ett sammansatt uttryck som följande:

>>>

```
>>> number = 5
>>> number >= 0 and number <= 10
True

>>> number = 42
>>> number >= 0 and number <= 10
False
```

I det här exemplet använder du `and` operatorn för att sammanfoga två jämförelseuttryck som låter dig ta reda på om `number` är i intervallet från `0` till `10`, båda inkluderade.

I Python kan du göra detta sammansatta uttryck mer kortfattat genom att kedja ihop jämförelseoperatorerna. Till exempel är följande kedjade uttryck ekvivalent med föregående förening:

>>>

```
>>> number = 5
>>> 0 <= number <= 10
True
```

Detta uttryck är mer kortfattat och läsbart än det ursprungliga uttrycket. Du kan snabbt inse att den här koden kontrollerar om numret är mellan `0` och `10`. Observera att i de flesta programmeringsspråk är detta kedjade uttryck inte meningsfullt. I Python fungerar det som en charm.

I andra programmeringsspråk skulle detta uttryck förmodligen börja med att utvärdera , `0 <= number` vilket är sant. Detta sanna värde skulle sedan jämföras med `10`, vilket inte är så vettigt, så uttrycket misslyckas.

Python bearbetar internt denna typ av uttryck som ett likvärdigt `and` uttryck, som `0 <= number and number <= 10`. Det är därför du får rätt resultat i exemplet ovan.

Villkorliga uttryck eller den ternära operatören

Python har vad den kallar villkorliga uttryck . Den här typen av uttryck är inspirerade av den ternära operatorn som ser ut `a ? b : c` och används i andra programmeringsspråk. Den här konstruktionen utvärderas till `b` om värdet av `a` är sant, och annars utvärderas till `c`. På grund av detta är ibland motsvarande Python-syntax också känd som den ternära operatorn.

Men i Python ser uttrycket mer läsbart ut:

```
variable = expression_1 if condition else expression_2
```

Detta uttryck returneras `expression_1` om villkoret är sant och `expression_2` inte. Observera att det här uttrycket är ekvivalent med ett reguljärt villkor som följande:

```
if condition:
    variable = expression_1
else:
    variable = expression_2
```

Så varför behöver Python denna syntax? PEP 308 introducerade villkorliga uttryck som ett försök att undvika förekomsten av felbenägna försök att uppnå samma effekt som en traditionell ternär operator som använder operatorerna `and` och `or` i ett uttryck som följande:

```
variable = condition and expression_1 or expression_2
```

Det här uttrycket fungerar dock inte som förväntat, och returnerar `expression_2` när `expression_1` är falskt.

Vissa Python-utvecklare skulle undvika syntaxen för villkorliga uttryck till förmån för en vanlig villkorssats. I vilket fall som helst kan den här syntaxen vara praktisk i vissa situationer eftersom den ger ett kortfattat verktyg för att skriva tvåvägsvillkor.

Här är ett exempel på hur du använder syntaxen för det villkorliga uttrycket i din kod:

>>>

```
>>> day = "Sunday"
>>> open_time = "11AM" if day == "Sunday" else "9AM"
>>> open_time
'11AM'

>>> day = "Monday"
>>> open_time = "11AM" if day == "Sunday" else "9AM"
>>> open_time
'9AM'
```

När `day` är lika med `"Sunday"`, villkoret är sant och du får det första uttrycket, , `"11AM"` som ett resultat. Om villkoret är falskt får du det andra uttrycket, `"9AM"`. Observera att på samma sätt som operatorerna `and` och `or` returnerar det villkorliga uttrycket värdet för ett av dess uttryck snarare än ett booleskt värde.

Identitetsoperatörer och uttryck i Python

Python tillhandahåller två operatorer `is` och `is not`, som låter dig avgöra om två operander har samma **identitet**. Med andra ord låter de dig kontrollera om operanderna refererar till samma objekt. Observera att identitet inte är samma sak som jämlikhet. Den senare syftar till att kontrollera om två operander innehåller samma data.

Här är en sammanfattning av Pythons identitetsoperatorer. Observera att `x` och `y` är variabler som pekar på objekt:

Operatör	Exempel på uttryck	Resultat
`is`	`x is y`	• `True` om `x` och `y` håll en referens till samma minnesobjekt
• `False` annars		
`is not`	`x is not y`	• `True` om `x` pekar på ett annat objekt än det som `y` pekar på
• `False` annars		

Dessa två Python-operatorer är nyckelord istället för udda symboler. Detta är en del av Pythons mål att gynna läsbarhet i sin syntax.

Här är ett exempel på två variabler `x` och `y`, som refererar till objekt som är lika men inte identiska:

>>>

```
>>> x = 1001
>>> y = 1001

>>> x == y
True

>>> x is y
False
```

I det här exemplet `x` och `y` hänvisar till objekt vars värde är `1001`. Så de är lika. De refererar dock inte till samma objekt. Det är därför `is` operatören återkommer `False`. Du kan kontrollera ett objekts identitet med den inbyggda `id()` funktionen:

>>>

```
>>> id(x)
4417772080

>>> id(y)
4417766416
```

Som du kan dra slutsatsen från `id()` resultatet, `x` och `y` har inte samma identitet. `x is y` Så de är olika objekt, och på grund av det returnerar uttrycket `False` . Med andra ord, du får `False` eftersom du har två olika instanser av `1001` lagrade i din dators minne.

När du gör en tilldelning som `y = x` , skapar Python en andra referens till samma objekt. Återigen kan du bekräfta det med funktionen `id()` eller `is` operatören:

>>>

```
>>> a = "Hello, Pythonista!"
>>> b = a

>>> id(a)
4417651936
>>> id(b)
4417651936

>>> a is b
True
```

I det här exemplet, `a` och `b` håll referenser till samma objekt, strängen `"Hello, Pythonista!"` . Därför `id()` returnerar funktionen samma identitet när du anropar den med `a` och `b` . På samma sätt `is` returnerar operatören `True` .

Slutligen `is not` är operatören motsatsen till `is` . Så du kan använda `is not` för att avgöra om två namn *inte* refererar till samma objekt:

>>>

```
>>> x = 1001
>>> y = 1001
>>> x is not y
True

>>> a = "Hello, Pythonista!"
>>> b = a
>>> a is not b
False
```

I det första exemplet, eftersom `x` och `y` peka på olika objekt i din dators minne, `is not` returnerar operatören `True`. I det andra exemplet, eftersom `a` och `b` är referenser till samma objekt, `is not` returnerar operatorn `False`.

Återigen `is not` lyfter operatören fram Pythons läsbarhetsmål. I allmänhet låter båda identitetsoperatörerna dig skriva checkar som läses som vanlig engelska.

Medlemskapsoperatörer och uttryck i Python

Ibland behöver du avgöra om ett värde finns i en containerdatatyp, till exempel en lista, tuppel eller uppsättning. Med andra ord kan du behöva kontrollera om ett givet värde *är* eller *inte är* medlem i en samling värden **.** Python kallar denna typ av kontroll för ett medlemstest .

Medlemskapstester är ganska vanliga och användbara i programmering. Som med många andra vanliga operationer har Python dedikerade operatörer för medlemskapstester. Tabellen nedan listar **medlemsoperatörerna** i Python:

Operatör	Exempel på uttryck	Resultat
`in`	`value in collection`	• `True` om `value` *finns* i `collection`
• `False` annars		
`not in`	`value not in collection`	• `True` om `value` *inte* finns i `collection` av värden
• `False` annars		

Som vanligt gynnar Python läsbarhet genom att använda engelska ord som operatorer istället för potentiellt förvirrande symboler eller kombinationer av symboler.

Python `in` och `not in` operatorerna är binära. Det betyder att du kan skapa medlemsuttryck genom att koppla två operander med endera operatorn. Men operanderna i ett medlemsuttryck har särskilda egenskaper:

- **Vänster operand** : Värdet som du vill leta efter i en samling värden
- **Höger operand** : Samlingen av värden där målvärdet kan hittas

För att bättre förstå `in` operatorn, nedan har du två demonstrationsexempel som består av att avgöra om ett värde finns i en lista:

>>>

```
>>> 5 in [2, 3, 5, 9, 7]
True

>>> 8 in [2, 3, 5, 9, 7]
False
```

Det första uttrycket returnerar `True` eftersom `5` finns i listan med nummer. Det andra uttrycket returnerar `False` eftersom `8` det inte finns i listan.

Medlemsoperatören `not in` kör det motsatta testet som `in` operatören. Det låter dig kontrollera om ett heltalsvärde *inte finns* i en samling värden:

>>>

```
>>> 5 not in [2, 3, 5, 9, 7]
False

>>> 8 not in [2, 3, 5, 9, 7]
True
```

I det första exemplet får du `False` eftersom `5` finns i mållistan. I det andra exemplet får du `True` eftersom `8` inte finns i värdelistan. Detta kan låta som en tungrodd på grund av den negativa logiken. För att undvika förvirring, kom ihåg att du försöker avgöra om värdet *inte är* en del av en given samling värden.

Sammankopplings- och upprepningsoperatorer och uttryck

Det finns två operatorer i Python som får en något annorlunda betydelse när du använder dem med sekvensdatatyper, som listor, tupler och strängar. Med dessa typer av operander `+` definierar operatorn en **sammanlänkningsoperator** och `*` operatorn representerar **upprepningsoperatorn** :

Operatör	Drift	Exempel på uttryck	Resultat
`+`	Sammankoppling	`seq_1 + seq_2`	En ny sekvens som innehåller alla objekt från båda operanderna
`*`	Upprepning	`seq * n`	En ny sekvens som innehåller föremål för `seq` upprepade `n` gånger

Båda operatorerna är binära. Sammankopplingsoperatorn tar två sekvenser som operander och returnerar en ny sekvens av samma typ. Upprepningsoperatorn tar en sekvens och ett heltal som operander. Liksom i vanlig multiplikation, ändrar inte ordningen på operanderna upprepningens resultat.

Här är några exempel på hur sammankopplingsoperatören fungerar i praktiken:

>>>

```
>>> "Hello, " + "World!"
'Hello, World!'

>>> ("A", "B", "C") + ("D", "E", "F")
('A', 'B', 'C', 'D', 'E', 'F')

>>> [0, 1, 2, 3] + [4, 5, 6]
[0, 1, 2, 3, 4, 5, 6]
```

I det första exemplet använder du sammanlänkningsoperatorn (`+`) för att sammanfoga två strängar. Operatören returnerar ett helt nytt strängobjekt som kombinerar de två ursprungliga strängarna.

I det andra exemplet sammanfogar du två bokstäver. Återigen returnerar operatören ett nytt tupelobjekt som innehåller alla objekt från de ursprungliga operanderna. I det sista exemplet gör du något liknande men den här gången med två listor.

När det kommer till upprepningsoperatorn är tanken att upprepa innehållet i en given sekvens ett visst antal gånger. Här är några exempel:

>>>

```
>>> "Hello" * 3
'HelloHelloHello'
>>> 3 * "World!"
'World!World!World!'

>>> ("A", "B", "C") * 3
('A', 'B', 'C', 'A', 'B', 'C', 'A', 'B', 'C')

>>> 3 * [1, 2, 3]
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

I det första exemplet använder du upprepningsoperatorn (`*`) för att upprepa `"Hello"` strängen tre gånger. I det andra exemplet ändrar du ordningen på operanderna genom att placera heltalstalet till vänster och målsträngen till höger. Det här exemplet visar att ordningen på operanderna inte påverkar resultatet.

I nästa exempel används repetitionsoperatorerna med en tupel respektive en lista. I båda fallen får du ett nytt objekt av samma typ som innehåller objekten i den ursprungliga sekvensen som upprepas tre gånger.

Valrossoperatören och uppdragsuttryck

Vanliga uppdragsutlåtanden hos `=` operatören har inget returvärde, som du redan har lärt dig. Istället skapar eller uppdaterar tilldelningsoperatören variabler. På grund av detta kan operatören inte vara en del av ett uttryck.

Sedan Python 3.8 har du tillgång till en ny operatör som möjliggör en ny typ av tilldelning. Den här nya tilldelningen kallas **tilldelningsuttryck** eller **namngiven uttryck** . Den nya operatorn kallas **valrossoperatorn** och det är kombinationen av ett kolon och ett likhetstecken ( ) `:=` .

Till skillnad från vanliga tilldelningar har tilldelningsuttryck ett returvärde, vilket är anledningen till att de är *uttryck* . Så operatören utför två uppgifter:

1. Returnerar uttryckets resultat
2. Tilldelar resultatet till en variabel

Valrossoperatorn är också en binär operator. Dess vänstra operand måste vara ett variabelnamn, och dess högra operand kan vara vilket Python-uttryck som helst. Operatören kommer att utvärdera uttrycket, tilldela dess värde till målvariabeln och returnera värdet.

Den allmänna syntaxen för ett tilldelningsuttryck är följande:

Det här uttrycket ser ut som en vanlig uppgift. Men istället för att använda tilldelningsoperatorn (`=`), använder den valrossoperatorn (`:=`). För att uttrycket ska fungera korrekt krävs i de flesta användningsfall de omslutande parenteserna. Men i vissa situationer behöver du dem inte. Hur som helst kommer de inte att skada dig, så det är säkert att använda dem.

Tilldelningsuttryck är praktiskt när du vill återanvända resultatet av ett uttryck eller en del av ett uttryck utan att använda en dedikerad tilldelning för att ta tag i detta värde i förväg. Det är särskilt användbart i samband med ett villkorligt uttalande. För att illustrera visar exemplet nedan en leksaksfunktion som kontrollerar längden på ett strängobjekt:

>>>

```
>>> def validate_length(string):
...     if (n := len(string)) < 8:
...         print(f"Length {n} is too short, needs at least 8")
...     else:
...         print(f"Length {n} is okay!")
...

>>> validate_length("Pythonista")
Length 10 is okay!

>>> validate_length("Python")
Length 6 is too short, needs at least 8
```

I det här exemplet använder du en villkorssats för att kontrollera om inmatningssträngen har färre än `8` tecken.

Tilldelningsuttrycket, `(n := len(string))` , beräknar stränglängden och tilldelar den till `n` . Sedan returnerar den värdet som blir resultatet av att anropa, `len()` som slutligen jämförs med `8` . På så sätt garanterar du att du har en referens till stränglängden att använda i vidare operationer.

Bitwise-operatorer och uttryck i Python

Bitvisa operatorer behandlar operander som sekvenser av binära siffror och arbetar på dem bit för bit. För närvarande stöder Python följande bitvisa operatorer:

Operatör	Drift	Exempel på uttryck	Resultat
`&`	Bitvis OCH	`a & b`	• Varje bitposition i resultatet är den logiska OCH för bitarna i motsvarande position för operanderna.
• `1` om båda bitarna är , `1` annars `0` .			
`	`	`	Bitvis ELLER	`a
• `1` om endera biten är `1` , annars `0` .			
`~`	Bitvis INTE	`~a`	• Varje bitposition i resultatet är den logiska negationen av biten i motsvarande position för operanden.
• `1` om biten är `0` och `0` om biten är `1` .			
`^`	Bitvis XOR (exklusiv ELLER)	`a ^ b`	• Varje bitposition i resultatet är den logiska **XOR** för bitarna i motsvarande position för operanderna.
• `1` om bitarna i operanderna är olika, `0` om de är lika.			
`>>`	Bitvis högerväxling	`a >> n`	Varje bit flyttas åt rätt `n` håll.
`<<`	Bitvis vänsterväxling	`a << n`	Varje bit flyttas åt vänster `n` .

Som du kan se i den här tabellen är de flesta bitvisa operatorer binära, vilket betyder att de förväntar sig två operander. Den bitvisa NOT-operatorn (`~`) är den enda unära operatorn eftersom den förväntar sig en enskild operand, som alltid ska visas till höger om uttrycket.

Du kan använda Pythons bitvisa operatorer för att manipulera dina data på dess mest granulära nivå, bitarna. Dessa operatorer är vanligtvis användbara när du vill skriva lågnivåalgoritmer, som komprimering, kryptering och andra.

Här är några exempel som illustrerar hur några av de bitvisa operatorerna fungerar i praktiken:

>>>

```
>>> # Bitwise AND
>>> #   0b1100    12
>>> # & 0b1010    10
>>> # --------
>>> # = 0b1000     8
>>> bin(0b1100 & 0b1010)
'0b1000'
>>> 12 & 10
8

>>> # Bitwise OR
>>> #   0b1100    12
>>> # | 0b1010    10
>>> # --------
>>> # = 0b1110    14
>>> bin(0b1100 | 0b1010)
'0b1110'
>>> 12 | 10
14
```

I det första exemplet använder du den bitvisa AND-operatorn. De kommenterade raderna börjar med `#` och ger en visuell representation av vad som händer på bitnivå. Notera hur varje bit i resultatet är den logiska OCH för bitarna i motsvarande position för operanderna.

Det andra exemplet visar hur den bitvisa OR-operatorn fungerar. I detta fall är de resulterande bitarna det logiska ELLER-testet av motsvarande bitar i operanderna.

I alla exemplen har du använt den inbyggda `bin()` funktionen för att visa resultatet som ett binärt objekt. Om du inte lindar uttrycket i ett anrop till `bin()` får du heltalsrepresentationen av utdata.

Operatörsprioritet i Python

Fram till denna punkt har du kodat exempeluttryck som oftast använder en eller två olika typer av operatorer. Men vad händer om du behöver skapa sammansatta uttryck som använder flera olika typer av operatorer, som jämförelse, aritmetik, Boolean och andra? Hur avgör Python vilken operation som körs först?

Tänk på följande matematiska uttryck:

Det kan finnas tvetydigheter i detta uttryck. Ska Python utföra additionen `20 + 4` först och sedan multiplicera resultatet med `10`? Ska Python köra multiplikationen `4 * 10` först och additionen sedan?

Eftersom resultatet är `60`, kan du dra slutsatsen att Python har valt det senare tillvägagångssättet. Om den hade valt det förra, skulle resultatet bli `240`. Detta följer en standard algebraisk regel som du hittar i praktiskt taget alla programmeringsspråk.

Alla operatörer som Python stöder har företräde jämfört med andra operatörer. Denna prioritet definierar i vilken ordning Python kör operatorerna i ett sammansatt uttryck.

I ett uttryck kör Python operatorerna med högsta prioritet först. Efter att ha erhållit dessa resultat, kör Python operatorerna med näst högsta prioritet. Denna process fortsätter tills uttrycket är helt utvärderat. Alla operatorer med samma prioritet utförs i ordning från vänster till höger.

Här är prioritetsordningen för Python-operatorerna som du har sett hittills, från högsta till lägsta:

Operatörer	**Beskrivning**
`**`	Exponentiering
`+x`, `-x`, `~x`	Enär positiv, unär negation, bitvis negation
`*`, `/`, `//`, `%`	Multiplikation, division, våningsindelning, modulo
`+`, `-`	Addition, subtraktion
`<<`, `>>`	Bitvisa skiftningar
`&`	Bitvis OCH
`^`	Bitvis XOR
`|`	`|`
`==`, `!=`, `<`, `<=`, `>`, `>=`, `is`, `is not`, `in`, `not in`	Jämförelser, identitet och medlemskap
`not`	Boolean INTE
`and`	Boolean OCH
`or`	Boolean ELLER
`:=`	Valross

Operatörer i toppen av tabellen har högst prioritet och de längst ner i tabellen har lägst prioritet. Alla operatorer i samma rad i tabellen har samma prioritet.

För att återgå till ditt ursprungliga exempel kör Python multiplikationen eftersom multiplikationsoperatorn har en högre prioritet än additionen.

Här är ett annat illustrativt exempel:

```
>>>
```

```
>>> 2 * 3 ** 4 * 5
810
```

I exemplet ovan höjer Python först `3` till makten , `4` vilket är lika med `81` . Sedan utför den multiplikationerna i ordning från vänster till höger: `2 * 81 = 162` och `162 * 5 = 810` .

Du kan åsidosätta standardoperatörens prioritet genom att använda parenteser för att gruppera termer som du gör i matematik. Underuttrycken inom parentes körs före uttryck som inte är inom parentes.

Här är några exempel som visar hur ett par parenteser kan påverka resultatet av ett uttryck:

>>>

```
>>> (20 + 4) * 10
240

>>> 2 * 3 ** (4 * 5)
6973568802
```

I det första exemplet beräknar Python uttrycket `20 + 4` först eftersom det är inslaget inom parentes. Sedan multiplicerar Python resultatet med `10` , och uttrycket returnerar `240` . Detta resultat är helt annorlunda än det du fick i början av det här avsnittet.

I det andra exemplet utvärderar Python `4 * 5` först. Sedan höjs det `3` till makten för det resulterande värdet. Slutligen multiplicerar Python resultatet med , `2` och returnerar `6973568802` .

Det är inget fel med att liberalt använda parenteser, även när de inte är nödvändiga för att ändra utvärderingsordningen. Ibland är det en bra praxis att använda parenteser eftersom de kan förbättra din kods läsbarhet och befria läsaren från att behöva återkalla operatörsföreträde från minnet.

Tänk på följande exempel:

Här är parenteserna onödiga, eftersom jämförelseoperatorerna har högre prioritet än `and` . Vissa kanske tycker att versionen inom parentes är tydligare än versionen utan parentes:

Å andra sidan kanske vissa utvecklare föredrar den här senare versionen av uttrycket. Det är en fråga om personlig preferens. Poängen är att du alltid kan använda parenteser om du känner att de gör din kod mer läsbar, även om de inte är nödvändiga för att ändra utvärderingsordningen.

Förstärkta uppdragsoperatörer och uttryck

Hittills har du lärt dig att ett enda likhetstecken (`=`) representerar tilldelningsoperatorn och låter dig tilldela ett värde till en variabel. Att ha en högerhandsoperand som innehåller andra variabler är helt giltigt, som du också har lärt dig. I synnerhet kan uttrycket till höger om tilldelningsoperatorn inkludera samma variabel som finns till vänster om operanden.

Den sista meningen kan låta förvirrande, så här är ett exempel som förtydligar poängen:

>>>

```
>>> total = 10
>>> total = total + 5
>>> total
15
```

I det här exemplet `total` är en **ackumulatorvariabel** som du använder för att *ackumulera* successiva värden. Du bör läsa det här exemplet som *`total` är lika med det aktuella värdet på `total` plus `5`* . Detta uttryck ökar effektivt värdet på , `total` vilket är nu `15` .

Observera att denna typ av tilldelning bara är meningsfull om variabeln i fråga redan har ett värde. Om du provar tilldelningen med en odefinierad variabel får du ett felmeddelande:

>>>

```
>>> count = count + 1
Traceback (most recent call last):
    ...
NameError: name 'count' is not defined. Did you mean: 'round'?
```

I det här exemplet `count` är variabeln inte definierad före tilldelningen, så den har inte ett aktuellt värde. Som en följd av detta tar Python upp ett `NameError` undantag för att informera dig om problemet.

Den här typen av tilldelning hjälper dig att skapa ackumulatorer och räknarvariabler, till exempel. Därför är det en ganska vanlig uppgift inom programmering. Som i många liknande fall erbjuder Python en mer bekväm lösning. Den stöder en stenografisyntax som kallas förstärkt tilldelning :

>>>

```
>>> total = 10
>>> total += 5
>>> total
15
```

På den markerade raden använder du operatorn augmented addition ( ) `+=` . Med den här operatorn skapar du en tilldelning som helt motsvarar `total = total + 5` .

Python stöder många utökade uppdragsoperatörer. Generellt sett ser syntaxen för den här typen av tilldelning ut ungefär så här:

Observera att dollartecknet (`$`) inte är en giltig Python-operator. I det här exemplet är det en platshållare för en generisk operatör. Ovanstående uttalande fungerar enligt följande:

1. Utvärdera `expression` för att producera ett värde.
2. Kör operationen som definierats av operatorn som föregår tilldelningsoperatorn (`=`), med det aktuella värdet av `variable` och returvärdet av `expression` som operander.
3. Tilldela det resulterande värdet tillbaka till `variable` .

Tabellen nedan visar en sammanfattning av de utökade operatorerna för aritmetiska operationer:

Operatör	Beskrivning	Exempel på uttryck	Motsvarande uttryck
`+=`	Lägger till den högra operanden till den vänstra operanden och lagrar resultatet i den vänstra operanden	`x += y`	`x = x + y`
`-=`	Subtraherar den högra operanden från den vänstra operanden och lagrar resultatet i den vänstra operanden	`x -= y`	`x = x - y`
`*=`	Multiplicerar den högra operanden med den vänstra operanden och lagrar resultatet i den vänstra operanden	`x *= y`	`x = x * y`
`/=`	Delar den vänstra operanden med den högra operanden och lagrar resultatet i den vänstra operanden	`x /= y`	`x = x / y`
`//=`	Utför våningsindelning av vänster operand med höger operand och lagrar resultatet i vänster operand	`x //= y`	`x = x // y`
`%=`	Hittar resten av att dividera den vänstra operanden med den högra operanden och lagrar resultatet i den vänstra operanden	`x %= y`	`x = x % y`
`**=`	Höjer den vänstra operanden till höger operand och lagrar resultatet i den vänstra operanden	`x **= y`	`x = x**y`

Som du kan dra slutsatsen från den här tabellen har alla aritmetiska operatorer en utökad version i Python. Du kan använda dessa utökade operatorer som en genväg när du skapar ackumulatorer, räknare och liknande objekt.

Såg de utökade aritmetiska operatorerna snygga och användbara ut för dig? Den goda nyheten är att det finns fler. Du har också utökade bitvisa operatorer i Python:

Operatör	Drift	Exempel	Likvärdig
`&=`	Förstärkt bitvis AND (konjunktion)	`x &= y`	`x = x & y`
`` ` ``	=`	Förstärkt bitvis ELLER (disjunktion)	`x
`^=`	Förstärkt bitvis XOR (exklusiv disjunktion)	`x ^= y`	`x = x ^ y`
`>>=`	Förstärkt bitvis högerväxling	`x >>= y`	`x = x >> y`
`<<=`	Förstärkt bitvis vänsterväxling	`x <<= y`	`x = x << y`

Slutligen har sammankopplings- och upprepningsoperatorerna också utökade variationer. Dessa varianter beter sig olika med föränderliga och oföränderliga datatyper:

Operatör	Beskrivning	Exempel
`+=`	• Kör en utökad sammanfogningsoperation på målsekvensen.	
• Föränderliga sekvenser uppdateras på plats.		
• Om sekvensen är oföränderlig skapas en ny sekvens som tilldelas tillbaka till målnamnet.	`seq_1 += seq_2`	
`*=`	• Lägger `seq` till sig själv `n` tider.	
• Föränderliga sekvenser uppdateras på plats.		
• Om sekvensen är oföränderlig skapas en ny sekvens som tilldelas tillbaka till målnamnet.	`seq *= n`	

Observera att operatorn för utökad sammanlänkning fungerar på två sekvenser, medan operatorn för utökad upprepning fungerar på en sekvens och ett heltal.

Slutsats

Nu vet du vilka **operatörer** Python stöder och hur man använder dem. Operatorer är symboler, kombinationer av symboler eller nyckelord som du kan använda tillsammans med Python-objekt för att bygga olika typer av **uttryck** och utföra beräkningar i din kod.

I den här handledningen har du lärt dig:

- Vad är Pythons **aritmetiska operatorer** och hur man använder dem i **aritmetiska uttryck**
- Vad är Pythons **jämförelse** , **booleska** , **identitet** , **medlemskapsoperatörer**
- Hur man skriver **uttryck** med operatorer för jämförelse, boolesk, identitet och medlemskap
- Vilka **bitvisa** operatorer Python stöder och hur man använder dem
- Hur man kombinerar och upprepar sekvenser med hjälp av **sammanlänknings-** och **upprepningsoperatorerna**
- Vad är de **utökade** uppdragsoperatörerna och hur de arbetar

Med andra ord, du har täckt oerhört mycket mark! Om du vill ha ett praktiskt fuskblad som kan dra ditt minne till allt du har lärt dig, klicka på länken nedan:

Med all denna kunskap om operatörer är du bättre förberedd som Python-utvecklare. Du kommer att kunna skriva bättre och mer robusta uttryck i din kod.