Innføring i Python

Steinar Knutsen

Historikk

- Først implementert av Guido van Rossum i 1991.
- Spin-off frå Amoeba, eit forskningsprosjekt innafor distribuerte system.
- Meint å fylle gapet mellom C og bash.
- Først implementert på Macintosh.
- Oppkalt etter Monty Python. `spam` og `bacon` er meir vanlege enn `foo` og `bar`.
- Stammer frå ABC, eit programmeringsspråk utvikla etter ein operasjonsanalyse av programmeringsprosessen. ABC var primært utvikla som undervisningsspråk.
- Påverka av Modula-3, C, etc.
- Finst tilgjengeleg på dei fleste plattformer.

- Vert i dag vedlikehalde av Guido van Rossum med fleire.
- Grunnkonsept:

  Objektorientert. Alt er objekt. Arv, inkludert multippel arv, er støtta i ein enkel modell.

  Dynamisk typa. Ein referanse har ikkje bunden type.

  Utvidbart. Veldokumentert, brukarvennleg API til C.

- Nyttige URLar:

  http://www.python.org/

  http://starship.python.net/

  http://www.stone-dead.asn.au/

Lister, for-løkker og if-setningar

```
languages = ["C", "Simula", "BCPL", "Intercal"]
OOlanguages = ["Simula", "Python", "Eiffel"]
nonOOlanguages = ["C", "Fortran-IV", "Algol-60"]
for lang in languages:
        if lang in OOlanguages:
                print lang, "er objektorientert."
        elif lang in nonOOlanguages:
                print lang, "er ikkje objektorientert."
        else:
                print lang, "er uklassifisert her."
```

Merk:

Innrykk markerer blokker. Lineskift er vanlegvis signifikant. Ein kan overstyre lineskift med `\`.

`for` itererer over ei liste, ikkje ein indeks.

`case`, som i td C, finst ikkje i Python.

Ein kan slette referansar med `del`.

Dictionaries, while og kommentarar.

```
langdict = {
        "Python": "Kjekt programmeringsspråk.",
        "Rexx": "Skripting på Amiga og stormaskiner frå IBM.",
        "Perl": "Programmeringsspråk oppfunne av ein SIL-lingvist."
}
aksess = 0
while aksess < 3:
        aksess = aksess + 1
        oppslag = raw_input()
        if oppslag == "avslutt":
                break
        elif oppslag == "Intercal": # Vi overser at folk vil lære om Intercal
                continue
        if langdict.has_key(oppslag):
                print langdict[oppslag]
        else:
```

```
                print "Ukjent språk."
else:
        print "Ein får maksimalt gjera tre oppslag i ordlista."
```

Merk:

`for` og `while` kan begge ha ein `else` som vert eksekvert om løkka ikkje vert avbrote vha `break`.

Python har ikkje kortformer av typen `i++`.

Liner har implisitt kontinuasjon om ein `{`, `(` eller `[` enno ikkje har vorte lukka.

Alle hashbare, eller statiske, objekt kan nyttast som nøkler i dictionaries.

`#` er kommentarmarkør.

Grunnleggande datatypar

Integer
: Heiltalsobjekt. Divisjon er avrunda mot $-\infty$. `-3/2` er mao `-2`.

Lange integer
: Som integer, men kan vera vilkårleg store. Dvs, ikkje avgrensa av MAXINT på systemet.

Flyttal
: Vanlegvis direkte implementert på toppen av double precision floats i C på den aktuelle plattforma.

Komplekse tal
: (x+yj)

Strengar
: Kan innehalde `\000`. Strengar er statiske objekt, ein kan *ikkje* endre ein streng i Python, slike effektar oppnår ein ved å konstruere nye strengar.

Lister
: «Dynamiske arrays.» Kan innehalde alle slags objekt. Treng ikkje innehalde berre ein type objekt i ei liste.

Tuples
: «Statiske arrays.» Fungerer på same måte som lister, men kan i likskap med strengar ikkje endrast.

Dictionaries
: Assosiative arrays. Kan nytte alle slags statiske, dvs hashbare, objekt som nøkkel. Tilsvarer ein «hash» i Perl.

Slice-notasjon

I strengar, lister og tuples, eller mao sekvensobjekt, starter nummereringa frå 0. Eit element vert adressert ved `objekt[indeks]`. Td: `"abc"[1]` returnerer `b`. Negative tal vert tolka slik at indeksen `-1` referer til det øvste elementet i sekvensen, ein tel baklengs nedover i sekvensen. `"abc"[-2]` returnerer `b`.

Alle sekvensobjekt stør slicenotasjon. Ein slice er ein subsekvens av ein sekvens. Notasjonen er `objekt[start:slutt]`. Ein nyttig måte å visualisere slices på er å sjå for seg at dei er mellom elementene i sekvensen, da dei fungerer som følger: 0 refererer til eit punkt før det første elementet i lista, 1 til eit punkt mellom det første og det andre, etc. Utelét ein `start` eller `slutt` vert dei henhaldsvis tolka som før starten på lista eller etter slutten av lista. Eksempelvis `"abcdef"[:3]` returnerer `"abc"`, medan `"abcdef"[2:4]` returnerer `"cd"`.

«Mutability» og referansar

Spissformulert har ikkje Python variablar, berre referansar. (Om ein tenker C kan ein sjå på som alt i Python er peikarar som alltid vert dereferert.) Python skil skarpt mellom «statiske» og «variable» objekt, om ein utfører ein operasjon på eit statisk objekt vil referansen verte endra til å peike på eit nytt objekt som svarer til den utførte operasjonen.

- Alle taltypar, strengar, funksjonar og tuples er statiske.
- Lister og dictionaries er variable.
- Brukardefinerte objekt kan fungere både som statiske og variable.
- Strengkonkatenering, som i `"SNOBOL" + "-IV"` vil dermed *ikkje* legge til `-IV` til `SNOBOL`, men derimot generere ein heilt ny streng. Tuples vil oppføre seg tilsvarande.

Viktige metodar for lister og dictionaries

Det viktigaste einskildpoenget er funksjonen `dir()`. `dir(x)` vil syne kva for metodar objekt gjer synlege for brukaren. Lister og dictionaries støre mange fleire metodar enn nemnt her, men her er nokre av dei aller viktigaste.

- `liste.append(x)` plasserer `x` på slutten av lista.
- `liste.sort()` sorterer lista.
- `liste.reverse()` reverserer lista.
- `liste.index(x)` returnerer indeksen til første instans av `x` i lista.
- `dictionary.has_key(x)` returner sant eller usant om `dictionary` henhaldsvis har eller ikkje har nøkkelen `x`.
- `dictionary.keys()` returnerer nøklane til `dictionary`.
- `dictionary.values()` returnerer verdiane lagra i `dictionary`.

- `dictionary.items()` returnerer parvis alle nøklar og verdiar i `dictionary`.
- `dictionary.get(key, x)` prøver å returnere `dictionary[key]`, om denne ikkje eksisterer vert `x` returnert.

Funksjonar og intervall og funksjonsmapping

```
def fact(n):
        if n == 0 or n == 1:
                return 1
        else:
                return n*fact(n-1)

print map(fact, range(10))
```

Merk:

`map` returnerer ei liste der `fact` har vorte køyrt med kvart element i lista generert av `range` som argument.

```
def fact2(n):
        if n == 0:
                return 1
        else:
                return reduce(lambda x,y:  x*y, range(1, n+1))

print filter(lambda x:  x > 5000, map(fact2, range(10)))
```

Merk:

`lambda` er kun syntaktisk sukker for å opprette funksjonsobjekt i Python.

Klasser, objekt og metodar

```
class eksempel:
        fotpistol = []
        def __init__(self, data):
                self.data = data
        def skyt(self):
                print self.fotpistol
        def lad(self, kule):
                self.fotpistol.append(kule)
ammunisjon = eksempel("Det meste går.")
fot = eksempel(42)
ammunisjon.lad(1)
fot.lad(2)
fot.skyt()
```

Merk:

`fot.skyt` sist i eksempelet vil skrive ut `[1, 2]` sia lister er variable objekt og metoden lad kun har

aksessert klasse-attributten `fotpistol`. Hadde definisjonen av `fotpistol` vorte flytta til `__init__` ville problemet ha vorte unngått.

Ein kan diverre ikkje arve frå typar i Python. (Det finst løysningar for dette, og problemet vil ikkje lenger vera aktuelt i Python 2.0.)

Ønsker ein å definere ei klasse som arver frå andre klasser: `class namn(Base1[, Base2, ...]):`. Søkerekkefølgen er «depth firsth, left to right.»

Operatoroverloading skjer ved å definere metodar med gitte namn, td `__add__`.

Namnerom og scoping

Python er «statisk skopa.» Alle symboloppslag vert gjort i følgande rekkefølge:

1. Lokalt namnerom. Symbol definert innafor same metode eller funksjon.
2. Globalt namnerom for den aktuelle modulen.
3. Innebygde funksjonar.

```
# Denne koden vil ikkje fungere
a = 12
def nokon_liker_basic():
        print a
        a = 0
```

Eksempelet ovanfor fungerer ikkje av di Python ser tilordninga i funksjonsdefinisjonen og går ut i frå at `a` er ein lokal variabel. Når funksjonen så prøver å skrive ut `a` før han er tilordna fører dette til ein feil. I tillegg vil heller ikkje den globale referansen `a` verte oppdatert til 0, sjølv om ein ikkje hadde hatt ein funksjon som førte

til at funksjonen ikkje vart ferdig utført før ein kom til tilordninga. Om ein faktisk *vil* gjera slike ting, må ein gjera som følgande:

```
a = 12
def nokon_liker_basic():
        global a
        print a
        a = 0
```

Derimot vil det følgande eksemplet oppføre seg som venta:

```
a = 12
def nokon_liker_basic():
        print a
```

Dvs, det skriv ut talet 12.

Exceptions

```
TeitException = "Brukardefinert meta-unntak."
try:  # Det er lov å nøste try-statements.
        try:
                raise TeitException
        finally:
                # Dette vert eksekvert uansett
                print "Her sett ein ofte clean-up kode."
except TeitException:
        # Dette vert eksekvert om det vert aktivert ein exception
        print "Denne teksten vil alltid verte skrive ut."
else:
        # Koden i else vert utfør om det ikkje vart aktivert
        # ein exception
        print "Denne teksten vert aldri skrive ut her."
```

Merk:

Ein kan spesifisere fleire ulike «exception handlers» i serie. Om ein ikkje opplyser nokon spesifikk exception, dvs `except:`, vil *alle* moglege exceptions verte fanga av den handleren. Om ein spesifiserer fleire ulike handlere, må difor ein slik settast sist.

Modular

```
import string
from sys import version
# version er ein streng som til dømes kan sjå slik ut:
# 1.5.2 (#1, Jun 11 1999, 02:41:53) [GCC 2.7.2.2+myc2]

print "Dette er Python, versjon", string.split(version)[0]
```

Merk:

`from modul import *` vil importere alle symbol frå `modul` inn i namnerommet der ein køyrer importeringa. Dette er ein effektiv måte å lage vanskeleg vedlikehaldbar kode.

Ein gjer ikkje noko spesielt for å lage ein modul. Ei fil med funksjonsdefinisjonar er allereie ein modul. `import kjekkefunksjonar` vil søke gjennom ein sti definert i `sys.path` på jakt etter fila `kjekkefunksjonar.py`.

I praksis?

- `open("filnamn", "modi")` opner filobjekt.
- `string` inneheld strengtenster.
- `re` inneheld Perl5-liknande regexp-funksjonalitet.
- `os` inneheld os-tenster som `fork` og `listdir`.
- `sys` inneheld systemavhengige data og ein del grensesnitt. mellom anna til systemvariablar.
- Denne lista bør avsluttast så fort som råd, da ho er *lang*.