



Norges teknisk-naturvitenskapelige
universitet
Institutt for datateknikk og
informasjonsvitenskap

TDT4102 Prosedyre-
og objektorientert
programmering
Vår 2016

Øving 1 (Python)

Frist: 2016-01-22

Mål for denne øvingen:

- Overføre programmeringskonsepter vi kjenner til fra Python til C++

Generelle krav:

- Bruk de eksakte navn og spesifikasjoner gitt i oppgaven.
- Det er valgfritt om du vil bruke en IDE (Visual Studio, XCode), men koden må være enkel å lese, compilere og kjøre.

Denne øvingen er tenkt som ekstra hjelp til de av dere som har noe programmeringserfaring fra før, men fra et annet språk enn C++. Gjennom enkle eksempler ønsker vi å vise likhetstrekkene mellom det du kan fra før og C++.

I denne øvingen skal vi demonstrere likheter og ulikheter mellom C++ og Python. Oppgavene vil i stor grad gå ut på å oversette Python-kode til C++, og kan kanskje være til hjelp som en slags oppfriskning av ITGK-kunnskapene. Samtidig vil de gi en mulighet til å overføre kunnskapen man har fra Python over til C++. Det vil også være viktig å lære seg å legge merke til de fallgruvene og forskjellene som finnes mellom de to språkene.

Merk at Python-versjonen som brukes i eksemplene er Python 3.4, som benyttes i IT grunnkurs. Dersom du er vant med Python 2.7 vil det være visse **forskjeller**, blant annet når det gjelder divisjon.

Lynintroduksjon til forskjellene mellom Python og C++

Generelt rammeverk

Siden vi ikke har gått så mye i dybden på C++ ennå, er det verdt å nevne enkelte ting som vi som regel trenger for å ha et gyldig program i C++. Kildekoden til et (nesten) minimalt C++-program, som ikke gjør noe som helst, ser slik ut:

```
1  #include <iostream>
2
3  int main() {
4      // Kode her
5      return 0;
6  }
```

En kort gjennomgang av hva som foregår her:

- Linje 1: Vi sier at vi trenger `iostream`-biblioteket, som blant annet inneholder funksjonalitet for utskrift til skjerm og input fra bruker. Et annet interessant bibliotek er f.eks. `cmath`, som inneholder sinus/cosinus/tangens, kvadratroter/potensfunksjoner og andre mattematiske funksjoner.
- Linje 3: Vi deklarerer `main`-funksjonen. Når du har kompilert og kjører programmet ditt vil det alltid starte med å utføre `main`-funksjonen. Legg merke til krøllparentesene, som markerer «kroppen» til funksjonen. Vi ser videre at `main`-funksjonen har *returtypen* `int`, som betyr at den kommer til å returnere et heltall. I C++ skal `main`-funksjonen alltid ha returtypen `int`.
- Linje 4: Denne linjen inneholder en kommentar. Disse starter med `//`, og sier at resten av linjen skal ignoreres. Som kommentaren sier er det her, mellom krøllparentesene, at koden til funksjonen skrives.
- Linje 5: Vi returnerer verdien 0 fra `main`. Dette er konvensjon for å si at programmet fullførte uten feil. I motsatt fall kan vi returnere verdien 1 for å indikere at programmet feilet. Legg merke til at verdien vi returnerer har samme type som vi deklarte at `main()` funksjonen skulle returnere.

Du legger kanskje merke til at linje 4 og 5 er *indentert*. I C++, som i de fleste programmeringsspråk, er det vanlig å indentere innholdet i *blokker*, det vil si alt som er mellom krøllparentesene etter en *if*-setning, funksjonsdeklarasjon, *for*-loop, og så videre. Dette gjøres for at kildekoden skal være oversiktlig, og slik at man lettere forstår hvilken kode som henger sammen. Hvor mye du vil indentere kan du velge selv – det er vanlig å bruke enten 4 mellomrom eller en tabulator (tegnet som blir lagt inn når du trykker **tab**-knappen i en vanlig teksteditor).

Utskrift til skjerm

I Python bruker man funksjonen `print(a,b,c,...)` for å skrive ut tekst til skjerm. I C++ sender vi i stedet tekststrenger og andre objekter vi ønsker å skrive til skjerm til objektet `std::cout` (standard utstrøm) ved å bruke `<<`-operatoren. Det er tillatt å nøste denne og dermed skrive ut flere objekter om gangen. Siste linje i figuren under gir et eksempel på det.

`std::endl` er et forhåndsdefinert objekt som representerer linjeskift. I Python bruker man `\n` i `print` for å eksplisitt skrive ut et linjeskift. Dette er også mulig å gjøre i C++: `\n` inni en tekststreng som sendes til `std::cout` vil gi et linjeskift, på lik linje som i Python.

I Python er `print` en innebygd funksjon som man kan bruke uten å importere noen biblioteker, mens for å bruke `std::endl` er man nødt til å inkludere `iostream`-biblioteket. Det er også verdt å si noe om navnet `std::endl`. Dette består av to deler, `std` og `endl`, adskilt av to kolon. Her er `endl` selve *funksjonsnavnet*, mens `std` er navnerommet (på engelsk *namespace*) som objektet `cout` befinner seg i. Et navnerom er en slags kategori av funksjoner som kan gå på tvers av ulike biblioteker. For at ikke navnene på funksjonene i standardbiblioteket skal kollidere med navnene på dine variabler og funksjoner, er alle funksjoner, variabler og objekter i standardbiblioteket plassert i navnerommet `std`.

Læreboken og foreleser vil av og til bruke en setningen `using namespace std` først i programmet for å slippe å skrive `std::` hver gang funksjoner fra standardbiblioteket brukes. Dette gjør programmene litt kortere, «penere» og lettere å lese. Dersom man bruker et annet bibliotek enn standardbiblioteket mye kan det også være nyttig å bruke `using namespace` for det biblioteket. Å benytte seg av `using namespace` er en grei praksis for små programmer, og de fleste vi skal jobbe med i dette faget. I større programsystemer unngås imidlertid ofte `using namespace` grunnet muligheten for navnekollisjoner, og mange programvareprosjekter og -bedrifter tillater ikke bruk av `using namespace`, deriblant [Google](#).

<code>print(2+3)</code>	<code>std::cout << 2+3 << std::endl;</code>
<code>print("Hello world!")</code>	<code>std::cout << "Hello world!" << std::endl;</code>
<code>print("2+5=", 2+5, sep='')</code>	<code>std::cout << "2+5=" << (2+5) << std::endl;</code>

(a) Python

(b) C++

Figur 1: Utskrift til skjerm

Variabler

I Python kan man introdusere variabler bare ved å tilordne et variabelnavn en verdi. I C++ må vi første gang en variabel brukes deklarerer den ved å skrive datatypen foran variabelnavnet. Det er vanlig å tilordne verdier til variabler samtidig som vi deklarerer dem, men det er ikke påbudt.

<code>a = 1</code>	<code>int a = 1;</code>
<code>b = 2</code>	<code>int b = 2;</code>
<code>c = a + b</code>	<code>int c = a + b;</code>
<code>d = c // b</code>	<code>int d = c / b;</code>
<code>print(d)</code>	<code>std::cout << d << std::endl;</code>
<code>d = d * b</code>	<code>d = d * b;</code>
<code>print(d)</code>	<code>std::cout << d << std::endl;</code>
<code>e = c</code>	<code>double e = c;</code>
<code>f = e / 2;</code>	<code>double f = e / 2.0;</code>
<code>print(f)</code>	<code>std::cout << f << std::endl;</code>
<code>g = c / b</code>	<code>double g = c / static_cast<double>(b);</code>
<code>print(g)</code>	<code>std::cout << g << std::endl;</code>

(a) Python

(b) C++

Figur 2: Variabeldeklarasjon og bruk

Øverst i Python-eksempelet ser vi at tallene 1 og 2 henholdsvis tilordnes variablene `a` og `b`. Både 1 og 2 er skrevet som heltall (ingen komma), og `a` og `b` vil derfor automatisk bli tolket som heltall av Python. I C++ kreves det at vi eksplisitt spesifiserer typen til variabelen i koden.

Vi har i dette eksempelet sett på to forskjellige datatyper, `int` og `double`. `int` er en type som kun kan representere heltall. En variabel av typen `int` tar på de fleste plattformer opp 32 bits i minnet, og kan dermed representere heltall mellom -2^{31} og $2^{31} - 1$. `double` er en type som inneholder *flyttall*, som brukes til å representere reelle tall. Ulikt en variabel av typen `int` er en variabel av typen `double` vanligvis 64 bits stor, og har en presisjon på mellom 15 og 17 sifre avhengig av størrelsen på tallet. En ting som er verdt å merke seg er at flyttall ikke kan representere alle reelle tall (da det er et uendelig antall av dem i alle intervaller), og uvetting bruk av flyttall kan dermed lett føre til «avrundingsfeil».

På linje 4 er det verdt å merke seg forskjellen mellom divisjonsoperatorene i Python og C++. I C++ gir den vanlige divisjonsoperatoren `/` *heltallsdivisjon* når den blir brukt til å dele to heltall. Dette betyr at svaret av divisjonen blir rundet ned til nærmeste heltall, og tilsvarer operatoren `//` i Python. Dersom man ønsker å få et desimaltall (flyttall) som svar fra en divisjon mellom to heltall, er man nødt til å *caste* det ene tallet til en flyttallstype, f.eks. `double`, og så utføre divisjonen. Å caste en variabel betyr å gjøre den om til en annen type. Caster man for eksempel heltallet 4 til `double`, vil man få flyttallet 4.0. Caster man et flyttall, for eksempel 4.5 til `int`, vil man få heltallet 4. Et eksempel på casting er vist til slutt i figuren over. Det finnes ulike former for casting, men foreløpig trenger du bare å bry deg om `static_cast`.

Input fra bruker

Python kan ta inn data fra brukeren ved hjelp av `input()`. I C++ bruker man i stedet objektet `std::cin`, som fungerer på en lignende måte som utskrift til skjerm med `std::cout`.

```
print("Skriv inn et tall:")
i = input()
print("Skriv inn et tall:")
j = input()
print("Summen av de to tallene:", float(i)+float(j))
```

(a) Python

```
double i;
double j;
std::cout << "Skriv inn et tall:" << std::endl;
std::cin >> i;
std::cout << "Skriv inn et tall:" << std::endl;
std::cin >> j;
std::cout << "Summen av de to tallene: " << (i+j) << std::endl;
```

(b) C++

Figur 3: Input fra bruker

I Python må vi konvertere inndataen til flyttall før vi kan legge dem sammen. Ved bruk av `>>`-operatoren tar C++ seg av å lese inn og tolke verdiens type. Dette fungerer slik at verdien som leses inn vil bli tolket som å ha samme type som variabelen den skal lagres i, altså `double` i eksempelet over.

If-setninger

En if-test ser slik ut i Python og C++:

```
b = 2
if b > 2:
    print ("B is greater than 2")
else:
    print ("B is less than or equal to 2")
```

(a) Python

```
int b = 2;
if (b > 2) {
    std::cout << "B is greater than 2" << std::endl;
} else {
    std::cout << "B is less than or equal to 2" << std::endl;
}
```

(b) C++

Figur 4: If-tester i Python og C++

En viktig forskjell mellom if-tester i Python og C++ er at Python ikke krever parenteser rundt det som skal testes. C++ krever dessuten krøllparenteser rundt kodeblokken som kjøres dersom if-testen er sann, og tilsvarende for else, i motsetning til i Python der indentering brukes til dette formålet.¹

¹Dersom kodeblokken kun er én linje lang kan krøllparentesene droppes. Imidlertid er dette en vanlig kilde til feil: hvis man legger flere linjer til kodeblokken, vil det uten krøllparenteser fortsatt kun være den første linjen som «omfattes» av testen, og de andre linjene vil bli kjørt uansett.

For-løkker

En enkel `for`-løkke som kjører fra 1 til 10 ser slik ut i Python og C++:

<pre>for i in range(1, 10+1): print (i)</pre>	<pre>for (int i = 1; i < 10+1; i++) { std::cout << i << std::endl; }</pre>
---	---

(a) Python

(b) C++

Figur 5: For-løkker i Python og C++

`for` ser litt annerledes ut, men vi finner igjen 1 og 10 som grenser også her. At vi har en steglengde på 1 er litt mindre intuitivt. Dette kommer av den siste delen av argumentet til `for`-løkken, `i++`, som tilsvarer `i = i + 1` eller i ord «øk i med 1». I eksempelet under brukes steglengde 2 i stedet, noe som fører til at kun oddetallene skrives ut:

<pre>for i in range(1,10+1,2): print (i)</pre>	<pre>for (int i = 1; i < 10+1; i = i + 2) { std::cout << i << std::endl; }</pre>
--	---

(a) Python

(b) C++

Figur 6: For-løkker med steglengde 2 i Python og C++

Uttrykk på formen `i = i + x` skrives ofte som `i += x` i C++. I vårt tilfelle kunne vi altså i stedet skrevet `i += 2`. Legg også merke til at vi uttrykker lengden på `for`-løkka vår som en sammenligning, `i <= 10`. Dette fungerer slik at `for`-løkken fortsetter så lenge denne sammenligningen er sann. Her kunne vi skrevet `i <= 10` i stedet for `i < 10+1`. Det er i C++ normalt å starte løkkene på 0 dermed og bruke `<` i stedet for `<=`, blant annet fordi tabeller i C++ har første indeks på 0, på samme måte som i Python.

While-løkker

<pre>i = 1 while i < 1000: i = i * 2 print(i)</pre>	<pre>int i = 1; while (i < 1000) { i = i*2; std::cout << i << std::endl; }</pre>
--	---

(a) Python

(b) C++

Figur 7: While-løkker i Python og C++

Igjen er det kun mindre forskjeller i syntaks mellom en `while`-løkke i Python og en i C++. Det er likevel greit å merke seg at C++ her også krever at det er parenteser rundt uttrykket som testes og at C++ bruker krøllparenteser for å markere koden som skal kjøres i løkken, i motsetning til i Python der indentering brukes til dette formålet.

Tabeller

<pre>t = [0]*10 for i in range(0,10): t[i] = i print (t[i])</pre>	<pre>int t[10]; for (int i = 0; i < 10; i++) { t[i] = i; std::cout << t[i] << std::endl; }</pre>
---	---

(a) Python

(b) C++

Figur 8: Tabeller i Python og C++

Tabeller opprettes på en litt annerledes måte i C++ enn i Python. For å opprette en tabell i C++ er vi (inntil videre) avhengig av å vite den endelige størrelsen før programmet kompileres, slik at det kan settes av nok minne. Når tabellen er opprettet vil den oppføre seg på nesten tilsvarende måte som i Python.

Funksjoner

<pre>def get_random_number(seed): return 4</pre>	<pre>int getRandomNumber(int seed) { return 4; }</pre>
--	--

(a) Python

(b) C++

Figur 9: Funksjoner i Python og C++

I Python defineres funksjoner ved å skrive `def funksjonsnavn(parametre)`. Dette er ulikt måten vi definerer funksjoner på i C++, der vi også må ha med *typen til funksjonens returverdi*. Som når variabler vanligvis defineres må funksjonens argumenter defineres med typer. I dette tilfellet ser vi at funksjonen tar inn et heltall, `int`.

En annen forskjell mellom Python og C++ er konvensjonen for funksjonsnavn. I Python bruker man stort sett understreker mellom de ulike «ordene» i et variabelnavn, for eksempel `get_random_number` fra eksempelet over. I C++ er det konvensjon å bruke såkalt «camelCase», for eksempel «getRandomNumber» i eksempelet.

Potens-operatoren

I Python bruker vi operatoren `**` for å opphøye et tall. I C++ finnes ikke denne, men vi har to andre alternativer. For enkle uttrykk, der eksponenten er et lite heltall, kan vi eksplisitt multiplisere tallet vi vil opphøye med seg selv, f.eks. $x^3 = x * x * x$. Alternativt kan vi inkludere biblioteket `cmath` og bruke funksjonen `std::pow`.


```
#include <iostream>
#include <cmath>

int main() {
    int fourSquared = std::pow(4,2);
    int fourCubed = std::pow(4,3);
    std::cout << "4^2: " << fourSquared
               << " 4^3: " << fourCubed << std::endl;
    return 0;
}
```

Figur 10: Eksempel på opphøying ved hjelp av `std::pow`

En ting å merke seg fra dette eksempelet er at uttrykket med `std::cout` går over to linjer. Dersom en linje ikke ender med semikolon, vil kompilatoren forsøke å tolke neste linje som en fortsettelse av det aktuelle uttrykket. Ofte separerer man logisk separate deler av et langt uttrykk på denne måten, slik at koden blir lettere å lese og linjene ikke blir for lange.

1 Kodeforståelse: oversett til Python (10%)

a) Oversett følgende kodesnutt til Python.

```
bool isFibonacciNumber(int n){
    int a = 0;
    int b = 1;
    while (b < n){
        int temp = b;
        b = a + b;
        a = temp;
    }
    if (b == n){
        return true;
    }
    else{
        return false;
    }
}
```

Figur 11: C++-kode

2 Oversett fra Python til C++ (90%)

Oversett følgende kodesnutter til C++, og sjekk at de både kompilerer og kjører i ditt IDE.

a) Fibonacci rekker

```
def fibonacci(n):
    a = 0
    b = 1
    print ("Fibonacci numbers:")
    for x in range(1,n):
        temp = b
        b = a + b
        a = temp
        print (x,b)
    print("-----")
    return b
```

Figur 12: Python-kode

b) Trekantall

```
def triangleNumbersBelow(n):  
    acc = 1  
    num = 2  
    print ("Triangle numbers below ", n, ":")  
    while acc+num < n:  
        acc = acc+num  
        num = num + 1  
        print (acc)  
    print()
```

Figur 13: Python-kode

```
def isTriangleNumber(number):  
    acc = 1  
    while (number > 0):  
        number = number - acc  
        acc = acc + 1  
    if number == 0:  
        return True  
    else:  
        return False
```

Figur 14: Python-kode

c) Sum av kvadrerte tall

```
def squareNumberSum(n):  
    totalSum = 0  
    for i in range(n):  
        totalSum += i*i  
        print (i*i)  
    print(totalSum)  
    return (totalSum)
```

Figur 15: Python-kode

d) Største av to tall

```
def max(a,b):  
    if a > b:  
        print ("A is greater than B")  
        return a  
    else:  
        print ("B is greater than or equal A")  
        return b
```

Figur 16: Python-kode

e) Primtall 1

```
def isPrime(n):  
    primeness = True  
    for j in range(2,n):  
        if n%j == 0:  
            primeness = False  
            break  
    return primeness
```

Figur 17: Python-kode

f) Primtall 2

```
def naivePrimeNumberSearch(n):  
    for number in range(2,n):  
        if isPrime(number):  
            print (number, " is a prime")
```

Figur 18: Python-kode

g) Største fellesnevner

```
def findGreatestDivisor(n):  
    for divisor in range(n-1,0,-1):  
        if n%divisor == 0:  
            return divisor
```

Figur 19: Python-kode

h) Telling med lister

```
def compareListOfNumbers(l):  
    r = [0]*3  
    for i in l:  
        if i < 0:  
            r[0] += 1  
        elif i == 0:  
            r[1] += 1  
        else:  
            r[2] += 1  
    print (r[0], " numbers were below zero")  
    print (r[1], " number were zero")  
    print (r[2], " numbers were greater than zero")
```

Figur 20: Python-kode