TDDE44. Programmering, grundkurs

Föreläsning 7

Jody Foo, jody.foo@liu.se

LINKÖPINGS UNIVERSITET

Föreläsningsöversikt

- Information om datorsalstentan
- Laboration 7
- Kort om algoritmer och komplexitet
- Skriva till fil
- Ännu mer om testning av kod
- Repetition

Laborationsassistent

- Var det roligt att programmera?
- Vill du fördjupa dina kunskaper?
- Vill du lära andra hur man programmerar?
- Sök timanställning eller amanuens:

  Timanställningar: https://liu.se/jobba-pa-liu/lediga-jobb?rmpage=job&rmjob=18964

  Amanuenser (proj.anst.): https://liu.se/jobba-pa-liu/lediga-jobb?rmpage=job&rmjob=18963

Datorsalstentan

logistik på plats, upplägg, inlämning

LINKÖPINGS UNIVERSITET

Datorsalstenta, 3 juni kl. 14.00-18.00

- Anmälan via Lisam eller LiU-appen.

  Anmälningsperiod: 4-24 maj

- Info på kurshemsidan under "Datorsalstenta"
- Kort allmän info

  https://www.ida.liu.se/edu/ugrad/datortenta/

Sal, inloggning, kontroll av bok

- Information om vilken sal du ska gå till vid SU17/18
- Inloggning

  Står "tentainloggning"

  Logga in med LiU-ID och ditt vanliga lösenord

  Liknande datormiljö som vid labbande, men utan internetåtkomst
- Kontroll av bok

Filer: tenta, datafiler, referens-PDF

- Din hemkatalog under tentan nås via `~/`
- Tentan som PDF samt eventuella filer som hör ihop med tentan finns i katalogen `~/given_files`
- `~/given_files` är skrivskyddad

Skriva tentan - rekommendation

- kopiera ev. datafiler från `given_files` till skrivbordet
- spara dina filer till skrivbordet
- Skrivbordet via terminal: `cd ~/Desktop`

Frågor under tentan

- **Praktiska frågor**: jour-assistenter
- **Frågor om tentauppgifter**: använd studentklienten
- **Studentklienten**: program som startas automatiskt vid inloggning och används för kommunikation med examinator samt inlämning av uppgifter

Tentaupplägg

- Två delar, Del 1 och Del 2
- Del 1: grundläggande uppgifter, samma storlek som Pythonuppgifter 1-3
- Del 2: något mer omfattande uppgifter

- Inga avdrag för avvikelser från PEP8 eller PEP257

Del 1

- Del 1 innehåller *fyra uppgifter*: 1.1, 1.2, 1.3, 1.4
- För att gå godkänt på Del 1 måste **alla uppgifter vara godkända**.
- För att få godkänt på en uppgift krävs 4 av max 6 poäng.

- Varje *uppgift* i Del 1 innehåller *tre deluppgifter*
- På varje *deluppgift* kan man få 0; 1 eller 2 poäng

Del 2

- Rättas ej om Del 1 inte är godkänd

- 0-4 poäng per uppgift (steg om 1 poäng)
- Uppgift 2.1 och 2.2 är av samma svårighetsgrad
- Uppgift 2.3 och 2.4 är av samma svårighetsgrad och svårare än 2.1 och 2.2

Betyg

- **Betyg 3**: Godkänd på del 1 + 3 poäng på del 2.
- **Betyg 4**: Godkänd på del 1 + 7 poäng på del 2.
- **Betyg 5**: Godkänd på del 1 + 11 poäng på del 2.

Inlämning av uppgifter

- En fil per uppgift, inklusive deluppgifter.
- Döp filen så att uppgiftsnummret finns med, t.ex. uppg11.py för ovanstående deluppgifter (står i tentan)
- Filen skickas in via studentklienten:
    1. Tryck på knappen "Skicka in uppgift"
    2. Välj vilken uppgift och välj vilken fil som skickas in
    3. Bekräftelse på att filen är mottagen "Begäran mottagen"

På kurshemsidan

- Exempeltenta och övningsuppgifter
- PDF som ni kommer ha tillgänglig som referens
- https://www.ida.liu.se/~TDDE44/tenta/

Laboration 7

Rättstavningskontroll

Laboration 7

- Uppgiften
  - Ordfrekvensdata
  - Redigeringsavstånd
  - Klasser
- Förberedelser: UML-diagram
- Problem som kan uppstå när man har mycket data

Laboration 7: Rättstavning

- "Fortsättning" på laboration 2
- Skript som använder ordfrekvensdata för att hitta okända ord i en text och föreslå alternativ till dessa ord med hjälp av redigeringsavstånd

  T.ex. hoppssn → hoppsan
- Rapport om okända ord + förslag sparas till fil

Laboration 7: filer

- **ordfrekvensfiler**

  innehåller antal förekomster av olika ord från diverse källor

  mest förekommande först i filerna

- **några textfiler**

  exempel på texter som ni kan "rättstava"

  använd gärna egna texter

Kortaste redigeringsavståndet

- Algoritm som räknar ut antal redigeringsoperationer som behövs för ordA → ordB
- Levenshtein-avstånd; *lägg till*, *ta bort*, *ersätt*

  "katt" → "hatt"; en operation (ersätt k med h)

  "katt" → "kalas"; tre operationer (ersätt första t:et med l, ersätt andra t:et med a, lägg till s)

  "hall → "hal"; en operation (ta bort ett l)
- Funktionen `minimum_edit_distance(s1, s2)` finns implementerad och redo att importera från filen `med.py`

Laboration 7: Klasser

- Rita UML-diagram och skriv ert program utifrån det.
- Uppdatera UML-diagrammet under tiden ni programmerar. Visas vid redovisning.
- Minst följande tre klasser:

  `SpellingWarning`: innehåller information om upptäckta ev. stavfel (som t.ex. plats, ordet och förslag)

  `Report`: innehåller `SpellingWarning`-objekt

  `WordFrequencyData`: innehåller ordfrekvensdata + metoder för att uppdatera/använda denna data

- För varje upptäckt fel skapar ni en instans av klassen `SpellingWarning`.
- Utöver dessa klasser är det upp till er.

Förberedelser innan ni börjar implementera

- Läs igenom instruktionerna på kurshemsidan
- Diskutera med er labpartner och rita upp UML-diagram.
- Visa upp och berätta hur ni tänkt för en assistent.

Tips

- En rapport har många upptäckta fel → jmf. flera hus i en stad
- Vid kontroll av flera filer → skapa flera rapporter (instanser)

- Behövs flera klasser? Vad vill ni representera? När ska man använda en inbyggd datatyp, t.ex. list eller dictionary och när ska man skapa en klass?
- Försök att se till så att en klass har ett ansvarsområde.
- Kom ihåg att metoder jobbar på data som klassen representerar.

Konsekvenser av öka mängden data att bearbeta

Laboration 7

- Körtid < 1 minut för minsta datafilen
- Går att köra <1 minut för största datafilen om man tillämpar vissa (rimliga) begränsningar.

Exempeluppgift

- `text.txt` innehåller en text på 1000 ord
- `ordfrekvenser.tsv` innehåller ordfrekvenser för 500k ord; varje rad består av
  - ett ord
  - ett tab-tecken (`'\t'`)
  - frekvensen som ordet förekommer i en viss text-korpus (samling av texter)
- Vi ska skriva en algoritm som tar fram vilka ord i `text.txt` finns med i `ordfrekvenser.tsv`

Loopar: Se upp för att göra om saker i onödan. Alt 1

- 1. **Kontrollera ett ord**. För varje `word_to_check` i texten som ska kontrolleras ← 1000

    1.1 **Läs in data**. läs in alla ordfrekevenser och spara dem som tupler av ord och frekvens i listan `freq_data` ← 500 000

    1.2 **Finns ord med i data?** Gå igenom *alla inlästa ord* och kontrollera om det är `word_to_check` ← 500 000

~ 1000 * (500 000 + 500 000) = $1* 10^9$ operationer
Om en operation tar 1 nanosekund ($1*10^{-9}$) tar vår algoritm 1s

Loopar: Se upp för att göra om saker i onödan. Alt 2

- 1. Läs in data. Läs in alla ordfrekevenser och spara dem som tupler av ord och frekvens i seprata listor för olika längd på orden (en lista per ordlängd)

500 000

- 2. för varje ord `word_to_check` i texten som ska kontrolleras

1000

2.1 gå igenom alla inlästa ord *av rätt längd* och kontrollera om det är `word_to_check`, avbryt så fort du hittar ordet

25 000 (antagande: snitt på 50 000 ord per ordlängd och att antal ord som tittas på är hälften i snitt)

500 000 + (1000 * 25 000) = 2,55 * 10^7 operationer
Om en operation tar 1 nanosekund (1*10^{-9}) tar detta alternativ 0,0255s

Loopar: Se upp för att göra saker i onödan

- Alt 1:

  Läser in all ordfrekvensdata för varje ord som ska kontrolleras

  Kontrollerar word_to_check mot alla ord, även de som inte har samma ordlängd.

- Alt 2:

  Läser in all ordfrekvensdata en gång.

  Kontrollerar word_to_check mot de ord som har samma längd

  Avbryter sökning när ordet har hittats

Komplexitet

Hur krävande är det att lösa en specifik uppgift med en specifik algoritm?

Vad är en algorithm?

- Informellt: en väldefinierad beräkningsbar (computational) procedur som givet ett värde, eller en mängd värden som input producerar ett värde, eller en mängd värden som output. (*Cormen et. al. 2009. Introduction to algorithms*)
- En algoritm sägs vara korrekt om det för varje korrekt indata, avslutar med korrekt utdata.

Kort om komplexitet

- Tids- och minneskomplexitet
- Hur mycket tid/minne behövs per input-enhet
- Dvs om vi skalar upp data, hur skalar resursanvändningen upp?

Komplexitet

- Hur mycket behöver datorn göra? Hur komplicerat är det att lösa detta problem, med den här algoritmen?
- Exempel, for-loop, leta efter största elementet

  Hur många operationer behöver algoritmen?

```
def get_max_value(values):
    max_value = values[0]
    for value in values:
        if value > max_value:
            max_value = value
    return max_value
```

Komplexitet

- Antal operationer är proportionerligt mot `len(values)`
- Ordo-notation (Big O notation)
- För nedanstående funktion:

  O(n), där *n* är storleken på input

```
def get_max_value(values):
    max_value = values[0]
    for value in values:
        # Ju större len(values) är, desto mindre påverkar
        # spelar antalet operationer i for-loopen roll.
       if value > max_value:
            max_value = value
    return max_value
```

Komplexitet, nästlade loopar

- Antalet par-kombinationer man kan bilda givet n värden
- [1, 2, 3 … n]

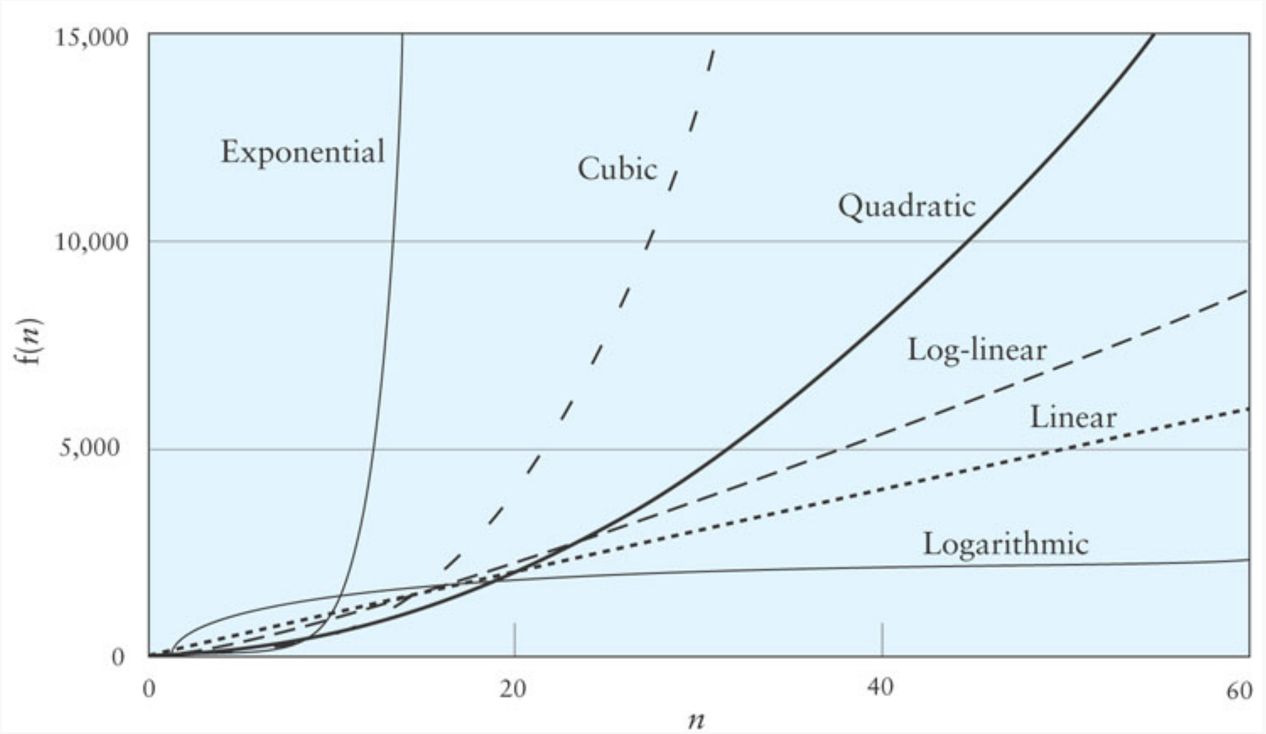
```
def get_pairs(values)
    pairs = []
    for v1 in values:
        for v2 in values:
            pairs.append(str(v1) + ":" + str(v2))
    return pairs
```

- $O(n^2)$

Komplexitet i stigande ordning

- O(1): konstant
- O(log n): logaritmisk
- O(n): linjär
- O(n^2): kvadratisk
- O(n^3): kubisk
- O(2^n): exponentiell

Komplexitet



Skriva till fil

Öppna fil för att läsa/skriva

- `open(filepath, mode)` returnerar den öppnade filen som ett file-objekt.
- mode är en sträng vars **defaultvärde** är `"r"` som i *"read"*
- andra giltiga värden på mode är `"w"` som i *"write"* och `"a"` som i *"append"*
- `mode='r'` **read**: vi kan endast läsa från filen
- `mode='w'` ***write***: vi kan endast skriva till filen, **existerande innehåll raderas**
- `mode='a'` **append**: vi kan endast skriva till filen, existerande innehåll sparas, **nytt innehåll läggs till slutet på filen**
- metoden `file.write(s)` skriver strängen `s` till filen `file`.

Öppna fil med explicit teckenkodning

- `open(filepath, mode, encoding='utf-8')` vi kan tvinga python att öppna filen med en viss teckenkodning genom att använda nyckelordsargumentet encoding. Behövs om man kör Windows och ska läsa/skriva textfiler kodade i utf-8

Exempel, skriva till fil

```
# öppna fil för att lägga till text till
datafile = open("filnamn.txt", "a")
datafile.write("Glöm inte bort")
datafile.write("radbryt!\n")

# stäng filen som vi öppnat
datafile.close()
```

Testning av kod

Testning jämfört med felsökning

Testning och felsöking

- **Felsökning**: Jag vet att något är fel med den här koden. Jag måste nu ta reda på var och varför det blir fel.
- **Testning**: Jag vet inte om programmet fungerar eller inte. Jag behöver ta reda på om alla delar fungerar som de ska.

Felsökning

- När vi vet att koden vi skrivit inte fungerar som den ska behöver vi t.ex. ta reda på
  - Under vilka omständigheter fungerar inte koden?
  - Var i koden uppstår felet? / Vad orsakar felet?
- Verktyg vi kan använda oss av vid felsökning
  - Felmeddelanden när programmet kraschar
  - Spårutskrifter
  - Olika debugverktyg

Testning

- Syftet med testning är att ta reda på om det vi testar fungerar eller inte. T.ex. om programmet uppför sig som vi förväntar oss.
- Stor bredd i det man kan referera till med "testning"
- Testning hjälpa oss påvisa förekomsten av fel.
- Rent praktiskt omöjligt att med hjälp av testning garantera frånvaron av fel, men vi kan använda testning för att minimera antalet fel!

Testning i denna kurs och utanför denna kurs

Testning i denna kurs

- Testning är en del av lärandemålen för kursen
- Ni har alla testat er kod på ett eller annat sätt

Testning inom mjukvaruutveckling

- Exempel på olika slags testning:

  enhetstestning - olika enheter, "units" av mjukvaran för sig själv, t.ex. enskilda funktioner, klasser

  integrationstest - test hur programmet fungerar när det interagerar med andra komponenter i ett system

  systemtest - test av systemet som helhet

  funktionstest - test av funktionaliteten hos ett program, t.ex. "lägga till en uppgift"

  prestandatest - test av prestanda hos programmet

- Speciella ramverk och program finns för att underlätta testning, men användning av dessa ligger utanför kursen.

Kan man (enhets-) testa allt?

- Man kan testa mycket, men för riktiga program är det svårt att testa allt.
- Exempel på svårare saker att testa:

  grafiska gränssnitt

  funktionalitet som är beroende av komplex data som är resultatet av interaktion mellan många komponenter i ett system

  funktionalitet som är beroende av yttre faktorer

Vad är omfånget av det vi testar i denna kurs?

- Test av funktioner, fungerar de som de ska?

  `funktion(indata)` → `returvärde`

- Enklare test av instanser av klasser, t.ex.

  `instans.metod()` → `returvärde`

  `instans.metod(indata)` → `returvärde`

  värdet hos instansvariabler efter att olika metoder körts

Vad ska tänka på vid testning?

- Båda nedanstående funktioner ska addera talen `v1`, och `v2` och sedan returnera resultatet.
- I nedanstående testkörning verkar båda vara korrekt implementerade?

```
def add_values1(v1, v2):
    return v1 + v2

def add_values2(v1, v2):
    return v1 * v2

# båda anropen returnerar 4, är båda implementationerna korrekta?
print(add_values1(2, 2))
print(add_values2(2, 2))
```

Vad behöver man veta för att testa sin kod?

Vad måste vi veta för att testa en enhet?

- Hur ser specifikationen ut? Dvs

- **Vilka starttillstånd och testförhållanden är giltiga?**

  Indata?

  Värden på instansvariabler innan en metod körs?

- **Vad ska enheten vi vill testa göra?**

  Hur ser utdata ut? Vilken utdata är giltig?

  Vilken utdata ska vi förväntas oss givet en viss indata?

Testdata

- I de flesta fall är det omöjligt att göra en uttömmande testning, dvs testa alla möjliga fall (t.ex. alla möjliga indata)
- Vi behöver titta på giltiga indata och försöka hitta *vilka kategorier av indata som finns*.
- Vilka indata ger är samma returvärde?
- Vilka indata testar samma aspekt av implementationen?
- Att hitta dessa kräver ofta att man förstår hur implementationen av en funktion eller klass kan göras.
- **Ett mål**: med så få test som möjligt kunna täcka så många fall som möjligt.

Vad behöver man göra för att testa sin kod?

Göra sitt program lättare att testa

- Genom att dela upp ett program i fler funktioner/klasser/metoder ökar man även i de flesta fall upplösningen på det som går att testa
- Funktioner som returnerar värden är lättare att testa än funktioner som inte returnerar värden.
- Genom att dokumentera sin kod finns det en specifikation som man kan använda som vägledning för att skapa rätt test.

Hur kan man testa?

- Att testa sitt program interaktivt kan ta tid (för att det sker manuellt), men ger stora möjligheter till att utforska resultaten.
- Att automatisera ett test, dvs skriva kod som testar funktionaliteten är snabbare och är upprepningsbart.

  Viktigt att veta vad vi ska testa!

Exempel på olika sätt att testa kod

Exempel på automatisering av test

- Automatisera genomförandet av testoperationen, t.ex.

  anrop av funktioner + okulär besiktning av resultaten för att validera korrekthet

- Exempel nedan där returvärden skrivs ut

```
def keep_even(values):
    output = []
    for value in values:
        if value % 2 == 0:
            output.append(value)
    return output

print(keep_even_values([]))
print(keep_even_values([1]))
print(keep_even_values([2]))
print(keep_even_values([1, 2, 3, 4]))
```

Exempel på automatisering av test

- Automatiserad validering

  automatisk jämförelse av faktiskt resultat av testoperationen med förväntat resultat

- Exempel med villkorssatser på nästa bild

Exempel på automatisering av test

```
def keep_even_values(values):
    output = []
    for value in values:
        if value % 2 == 0:
            output.append(value)
    return output

def test_keep_even_values():
    if keep_even_values([]) != []:
        return False
    if keep_even_values([1]) != []:
        return False
    if keep_even_values([2]) != [2]:
        return False
    if keep_even_values([1, 2, 3, 4]) != [2, 4]:
        return False
    return True

if test_keep_even_values():
    print("keep_even_values() fungerar.")
else:
    print("keep_even_values() fungerar INTE.")
```

Nyckelordet assert

- Nyckelordet `assert` används när man vill kontrollera om ett uttryck är sant i syfte att fånga upp fall när det inte är det.
- Om uttrycket är sant går Python vidare i koden.
- Om uttrycket är falskt, inträffar ett `AssertionError` som sedan kan fångas.
- Man kan använda villkorssatser i samma syfte, men koden kan bli kortare och tydligare med `assert`

- Syntax: `assert` *`uttryck`*

Exempel på automatisering av test (med assert)

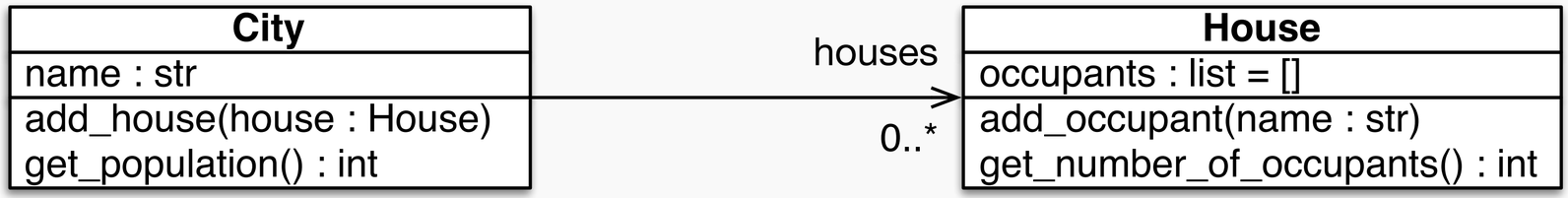
```
def keep_even_values(values):
    output = []
    for value in values:
        if value % 2 == 0:
            output.append(value)
    return output

def test_keep_even_values():
    try:
        assert keep_even_values([]) == []
        assert keep_even_values([1]) == []
        assert keep_even_values([2]) == [2]
        assert keep_even_values([1, 2, 3, 4]) == [2, 4]
        return True
    except AssertionError:
        return False

if test_keep_even_values():
    print("keep_even_values() fungerar.")
else:
    print("keep_even_values() fungerar INTE.")
```

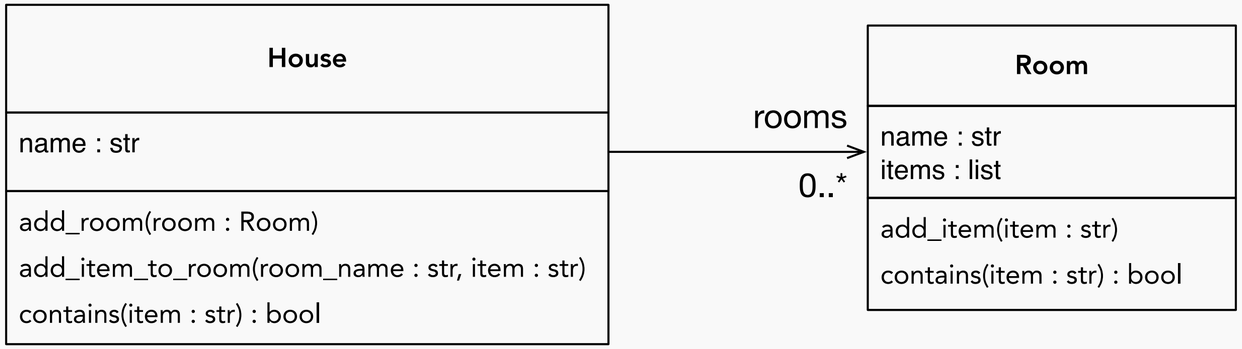
Repetition

Exempel på aggregering



Association och scope

- Associationspil implementeras som en variabel i "från"-klassen.
- Om fler än 1 referens ska finnas, använd t.ex. lista eller dictionary
- Samma variabelnamn och metodnamn kan användas i flera klasser.



Vad händer när vi skapar en instans av en klass?

- Skapa instans av klass: `Klassnamn(arg1, arg2 ... argn)`
- Python skapar ett objekt och kör klassens `__init__()`-metod
- Objektet som skapats skickas som det första argumentet till `__init__()`.
- Argument som skickats med vid anrop av klassnamnet skickas vidare till `__init__()` efter `self`.

Hur kan nedanstående tolkas - syntax är viktigt

1. `a`
2. `a()`
3. `a(b)`
4. `a[b]`
5. `a.b`
6. `a.b()`
7. `a.b = c`
8. `a = b.c`
9. `a = b.c()`

Beståndsdelar i ett program

- **värden (*value*)**: t.ex. siffror eller text (kallas för strängar)
- **operatorer (*operators*)**: token/symbol som står för en instruktion
- **variabler (*variable*)**: namngivna referenser till värden
- **uttryck (*expression*)**: ett uttryck kan beräknas till ett värde (ibland säger man även utvärderas istället för beräknas)
- **nyckelord (*keywords*)**: speciella ord som refererar till instruktioner som ska utföras
- **skiljetecken (*punctuators*)**
- **sats (*statement*)**: fullständig instruktion