

TDDE24

Information inför tentan

Jonas Kvarnström

■ Innehåll

- När, var, hur?
 - Förberedelser
 - Krav på korrekta lösningar
 - Uppgiftsval och lösningsordning
 - Att förstå en uppgift och testa lösningen
 - Fallgropar
 - Förändringar i år
- (På seminarierna: Genomgång av exempeltenta och konkreta uppgifter med seminariehandledarna)

**Fråga gärna
under tiden!**

Allmän info om tentor – läs på själva!

Länkar:

<https://www.ida.liu.se/~TDDE24/tenta/index.shtml>

När och var?

Tentamenstillfällen

LIU - [Student](#) - Tentamenstillfällen



Utb. kod	Kursnamn	Datum	Tid	Ort	Lokal / Anmälningssperiod
TDDE24/DAT1	<i>Funktionell och imperativ programmering, del 2</i>	2025-01-14	14 - 19	Linköping	2024-12-15 - 2025-01-04

När? (2)

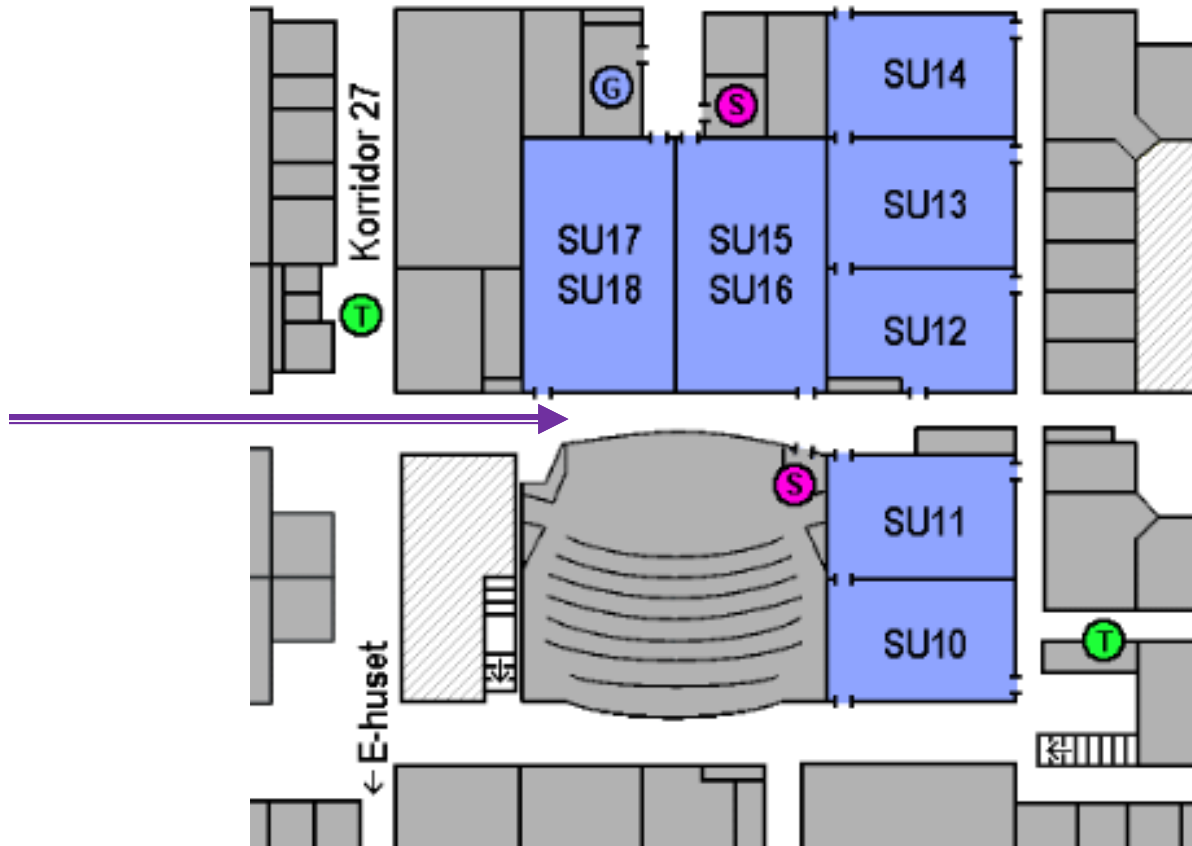
- 2024-12-02 läsperiod vecka 5
- 2024-12-03 läsperiod
- 2024-12-04 läsperiod
- 2024-12-05 läsperiod
- 2024-12-06 läsperiod
- 2024-12-07
- 2024-12-08
- 2024-12-09 läsperiod vecka 6
- 2024-12-10 läsperiod
- 2024-12-11 läsperiod
- 2024-12-12 läsperiod
- 2024-12-13 läsperiod
- 2024-12-14
- 2024-12-15
- 2024-12-16 läsperiod vecka 7
- 2024-12-17 läsperiod
- 2024-12-18 läsperiod
- 2024-12-19 läsperiod
- 2024-12-20 läsperiod
- 2024-12-21
- 2024-12-22

**Inte "lov",
utan
egna studier
inför tentor!**

- 2024-12-23 självstudier vecka 8
- 2024-12-24
- 2024-12-25
- 2024-12-26
- 2024-12-27 självstudier
- 2024-12-28
- 2024-12-29
- 2024-12-30 självstudier vecka 9
- 2024-12-31 självstudier
- 2025-01-01
- 2025-01-02 självstudier
- 2025-01-03 självstudier
- 2025-01-04
- 2025-01-05
- 2025-01-06
- 2025-01-07 omtentor eller självstudier v10
- 2025-01-08 omtentor eller självstudier
- 2025-01-09 omtentor eller självstudier
- 2025-01-10 omtentor eller självstudier
- 2025-01-11
- 2025-01-12
- 2025-01-13 tentor
- 2025-01-14 tentor – **TDDE24!**

■ Datortentamen

- På universitetet i SU-salar
- Tentavakter sitter utanför SU17/18
 - Gå dit i god tid, bli anvisad en plats



Hur?

- Programmera på **nedlåst dator**
 - Kommer åt <https://docs.python.org/3.11/> men inte annat på nätet
 - Speciell ikon för att starta webbläsare; se instruktioner

Python 3.11.6 documentation

Welcome! This is the official documentation for Python 3.11.6.

Parts of the documentation:

What's new in Python 3.11?

or all "What's new" documents since 2.0

Tutorial

start here

Library Reference

keep this under your pillow

Language Reference

describes syntax and language elements

Python Setup and Usage

how to use Python on different platforms

Python HOWTOs

in-depth documents on specific topics

Installing Python Modules

installing from the Python Package Index & other sources

Distributing Python Modules

publishing modules for installation by others

Extending and Embedding

tutorial for C/C++ programmers

Python/C API

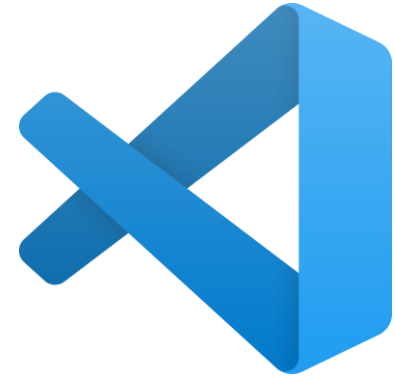
reference for C/C++ programmers

FAQs

frequently asked questions (with answers!)

- Programmera på **nedlåst dator**

- Kan använda **emacs** och liknande, men även **vscode**
 - Bara grunderna
 - Inga extrapaket / plugins – vi ska testa **era** kunskaper, inte utvecklingsmiljöns automatiska förslag
 - Konsekvens: Exempelvis ingen integrerad debugger



**Testa gärna att använda en "ren" vscode utan plugins
Se till att ni kan använda terminalen vid behov**

- Kan använda **standardpaket** från Python
 - Inte t.ex. Numpy, ...

- Att stänga av extensions (t.ex. Python-utökningar):
 - Kommandoraden: **code --disable-extensions**
 - Eller <https://code.visualstudio.com/docs/editor/extension-marketplace>

Disable an extension

If you don't want to permanently remove an extension, you can instead temporarily disable the extension by clicking the gear button at the right of an extension entry. You can disable an extension globally or just for your current Workspace. You will be prompted to reload VS Code after you disable an extension.

If you want to quickly disable all installed extensions, there is a **Disable All Installed Extensions** command in the **Command Palette** and **More Actions** (**...**) dropdown menu.

Extensions remain disabled for all VS Code sessions until you re-enable them.

Hur? (3)

- **Lämna in** via tentaklienten:

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

AES - Student Client (on tlhw-3-4)

Student Information:		Course Information:		Exam:	
Name:	John Doe	Course Code:	732A62	Start Time:	2019-10-
Person Number:	121212-121	Course Name:	Time Seri	Server Connection:	Connecte
Client ID:	SC35003	Course Language:	English		

Unread Messages:

Time	From	To	Regarding	Subject
------	------	----	-----------	---------

Read Messages:

Time	From	To	Regarding	Subject
------	------	----	-----------	---------

Grading Information:

Exam Grade: Set after exam (2019-10-16 10:45)
Assignment #1: Assessed after exam (2019-10-16 10:45)
Assignment #2: Assessed after exam (2019-10-16 10:45)

Send Question Send Assignment

■ Att lämna in:

- Varje **numrerad** uppgift är en egen inlämning
 - Men t.ex. 2a och 2b är samma inlämning, samma fil
- Testa inlämning redan efter första lösningen!
 - Se att du förstår hur det fungerar
 - Se efter att du får **bekräftelse** från systemet
 - Vill du ändra dig kan du byta ut en inlämning mot en ny
- Lämna in alla uppgifter i tid!
 - När tentan avslutas stängs systemet
- Vi granskar inga uppgifter under tentan – allt sker efteråt!

- **Skicka tentafrågor** till examinatorn via tentaklienten
 - Besvaras så snabbt vi hinner
 - Kan dröja – vid ”vanliga” tentor tar någon bara emot frågor vid 1-2 tillfällen
 - Läs genom allt tidigt
 - Spara inte frågorna till slutet!
- **Tekniska frågor?** Räck upp handen istället!
 - Problem med tangentbord/mus/dator
 - Kan inte spara fil / starta program
 - Problem med tentaklienten
 - ...

- Examinatorn kan **skicka ut information** via tentaklienten!
 - Förtydliganden vid vanliga missförstånd, ...

Tentaklienten har inga popups för notifikationer!

Titta själv i meddelandelistan då och då

Unread Messages:				
Time	From	To	Regarding	Subject

Förberedelser...

- Vill du skriva tentan? **Anmäl dig!**
 - Det finns **ingen** möjlighet för oss att ta emot studenter i efterhand
 - **Dubbelkolla** din anmälan

Vi bestämmer inte över detta
Vi kan inte ”vara snälla”
LiU-regler, central hantering!

- **Läs** instruktioner, när de är uppdaterade för 2024:
- <https://www.ida.liu.se/~TDDE24/tenta/index.shtml>
- Tentans försättsblad, med detaljerad information, kommer upp *ett par dagar* innan tentan
 - Skickas *kanske* ut via epost, men problem med regler kring anonymitet
- Även länkar på den sidan

Information om tentan

Information om tentan ges på flera sätt:

- ▶ **Allmän information om tentor på LiU**
- ▶ **Allmän information om datortentamen på IDA**, specifikt **info om tentaklienten**
- ▶ **Bilder från föreläsning/seminarium hösten 2022** med information om tentan i TDDE24
- ▶ **Försättsblad / instruktioner** till tentan 2023-01-10. Nya instruktioner kommer till varje ny tenta.
- ▶ **Ett seminarium** ger en möjlighet att gå genom gamla tentauppgifter. Ni kan också ställa frågor, men tänk på att seminarieledarna kanske inte kan svara på alla detaljfrågor.

- Gör labbarna klara – de är en bra förberedelse
- Titta gärna på gamla tentor
 - <https://www.ida.liu.se/~TDDE24/tenta/index.shtml>

Men uppgifterna behöver inte alls se ut som i tidigare tentor!

**Läs studiematerialet några gånger till
Fastna inte i gamla uppgifter
Vi examinerar generella kunskaper**

- Öva på programmering – i allmänhet

- Labbar vs tenta:
 - På labbar får man tid för att lära sig, prova sig fram
 - På tentan ska man redan ha kunskapen, inte behöva prova
 - Testar delvis att man *inte längre behöver slå upp lösningar* utan kan hitta rätt lösningar *på egen hand och snabbare*

Kursens innehåll / studiematerial

Testas av labbar

Testas av tenta

Mall för Python-filer

- Vi ger er en fil för varje uppgift, en "mall"...
- Använd den på rätt sätt!

Här lägger du din egen kod för uppgift 1.
Docstrings är inte ett krav, men funktionerna behöver vara
lättförståeliga och kan behöva kommentarer av den anledningen.

```
def datause(seq: list):  
    pass
```

Här skriver du din egen kod
Ersätt "pass" med din kod
(eller fortsätt efter "pass")

```
def check_python_version():  
    ...vår hjälpkod, kan ignoreras...  
    ...kan också tänkas ändras inför årets tentor...
```

Lägg inte dubbla definitioner
(både vår 'def' och en egen)

Fortsättning följer i nästa bild...

```
def run_tests():
```

```
    # De här testerna står uttryckligen som assertions i uppgiften på tentan.
```

```
    print("Kör uppgiftens tester...")
```

```
    assert datause([12, None]) == [12, 0, 12]
```

```
    assert datause([1, 5, 17, 2]) == [25, 25]
```

```
    assert datause([None, 1, 12, 5, None, 22, 21, None, 2]) == [0, 18, 43, 2, 43]
```

```
    # Här lägger du dina egna tester. Du kan till exempel skapa egna
```

```
    # assertions, eller lägga till andra tester såsom enkla utskrifter
```

```
    # av resultatet av en körning.
```

```
    print("Kör egna tester...")
```

```
    print("Kör utskriftstester...")
```

```
    # Här kommer några tester där vi kan
```

```
    # ändå kan skriva ut resultatet.
```

```
    print("Resultat 1:", datause([12, 34.5, 67]))
```

```
    print("Har kört alla tester")
```

**Egna tester inuti run_tests(),
inte utanför funktionen!
Annars körs de även vid rättning
→ kan krascha systemet**

```
if __name__ == '__main__':  
    # Denna kod körs när någon kör 'python3.11 ex1.py' från kommandoraden.  
    # Den körs INTE vid 'import ex1' -- för att testerna inte ska köras  
    # vid rättningen då vi istället kör våra egna tester.  
    check_python_version()  
    run_tests()
```

Krav på korrekta lösningar

- En korrekt lösning:
 - Fungerar exakt som beskrivningen säger
- Om lösningen är felaktig på något sätt, ger några felaktiga svar:
 - Ju **tydligare tankegång**, desto bättre
 - **Tydlig kod, tydliga kommentarer**, tydligt hur det var **tänkt**,
tydligt *var det gick fel*
 - Visar **förståelse** för problemet och lösningen,
och **demonstrerar** för granskaren att du hade den förståelsen:
Det man inte visar upp kan man inte få poäng för...
 - Avdrag **även om lösningen är uppenbar** för granskaren
 - Inlämningen är ett slutligt svar,
inte ett mellansteg som granskaren ska arbeta med

För många problem?

- **Rätt på 50%** av testfallen? Ger **inte** automatiskt 50% av poängen!
 - 50% av testfallen kan vara ***mycket enklare***
 - Programmering kräver ***precision***
 - Det är ***väldigt*** mycket lättare att få ***rätt svar ibland*** än att verkligen ***programmera korrekt!***
- Istället: Varje problem bedöms separat
 - Var det ett **enkelt** programmeringsfel, som ett "stavfel"?
 - Uppenbart **missförstånd** av uppgiften?
 - Gör missförståndet att man undvek uppgiftens stora poäng, eller en av de större svårigheterna? Eller påverkar det egentligen ingenting?
 - På väg åt helt **fel håll**?
 - **Svårförståelig lösningsgång** som inte förklaras?
 - ...
 - ...

Kommentera hur du tänker!

- Lösningar är **väldigt** olika, ofta **inte** uppenbara för läsaren
 - Kommentera en aning, så ökar chansen att läsaren förstår,
 - Kan minska risken att få 0 poäng för felaktig och oförståelig lösning, om:
 - Tanken bakom är korrekt
 - Kommentaren gör felet tillräckligt uppenbart
- Kan kommentera på svenska eller engelska

**Vi ser (oftast) vad koden gör –
men varför gör den det?**

- Allmänna krav:
 - Lämna in **körbar kod, körbara filer**
 - Ska gå att köra även efter den där sista finputsningen...
 - Har du egna testfall som misslyckas?
Alla testfall ska köras i `run_tests()`

- Allmänna krav:
 - Använd de givna filerna och döp inte om dem (ex1.py, ex2.py, ...)
 - Vår granskning använder hjälpscript, som kan missa inlämningar med felaktiga namn
 - Byt inte namn på funktionerna
 - Då hittas de inte av våra tester...
 - Skriv kod som är **tillräckligt välstrukturerad och väldokumenterad** för att en granskare **enkelt ska förstå** den (särskilt viktigt om lösningen inte är korrekt...)

■ Icke-krav:

- Kodan behöver **inte** fungera för **felaktiga indata** (se uppgiften!)
 - ”Skriv en funktion som tar ett heltal x och ...”
 - Skickar någon in ett flyttal, eller en sträng, eller None, får vad som helst hända
 - Ingen sådan felkontroll behövs
- Man får bryta mot ”ytliga” kodningsstandarder
 - Mellanrum, indentering, radlängd, blankrader
 - Namngivning som snake_case eller camelCase eller ...
 - Men koden ska fortfarande vara lättläst och lättförståelig

Uppgiftsval och lösningsordning

- **Skumma genom** uppgifterna först
 - Vilka tycker **du** verkar lättast?
 - Börja gärna där!

- Tänk på:
 - Det kan finnas **deluppgifter** som är mycket lättare än helheten
 - Ge inte upp om *hela* uppgiften verkar för svår

 - ***Långa instruktioner*** är inte samma sak som ***svår uppgift***
 - Instruktioner kan vara *långa och tydliga*

Lösningsordning: Byta ordning?



- Om du **kör fast** helt och hållet:
 - Kanske dags att byta uppgift... efter ett tag
- Men försök att **inte hoppa för mycket** fram och tillbaka
 - Behöver komma till slutet på några uppgifter

Lösningsordning: Lösa alla eller vissa?

- Frestande att **lösa så många uppgifter som möjligt**
 - Men vad hjälper det, om man tappar många poäng på slarvfel?
- **Många** skulle tjäna på att undersöka **om svaren är korrekta**
 - **Läs** uppgiften en gång till (eller två), kolla villkoren mot koden och resultaten (inte ovanligt med fel som uppenbart bryter mot instruktionerna)
 - **Testa**, skapa fler testfall (mer om det senare)
 - Många fel var egentligen **ganska lätta att upptäcka** och **väldigt lätta att fixa**
 - Men ger ändå avdrag – om det var lätt att upptäcka och fixa, *borde det ha fixats*

Att förstå en uppgift

- **Förstå uppgiften** och **utnyttja ledtrådar**
 - Läs *noga*, läs *flera gånger* – programmering kräver **precision**
 - Försök se alla **detaljer** och **ledtrådar**
 - Försök att undvika **missförstånd** – måste lösa **rätt uppgift**

**Kan inte vara
negativa**

Implementera därför en funktion `oval(w,h)` där parametrarna är två heltal $w \geq 0$ och $h \geq 0$ motsvarande bredden och höjden på den önskade ovalen. Funktionen ska **returnera en lista** med h element ("rader") indexerade $0..h-1$, där varje element i sin tur är en lista med w element indexerade $0..w-1$ ("kolumner") som är strängen 'X' om punkten ligger *inuti ovalen* eller strängen '.' om punkten ligger *utanför ovalen*. Exempelvis ska `oval(4,3)` ge:

Ska ha h element...

$h=0 \rightarrow$ lista [] med 0 element, inget annat, oavsett w !

Underlistor ska ha w element...

$h=1, w=0 \rightarrow [[]]$

$h=0, w=12 \rightarrow []$

Förståelse: Ibland finns det tips...

Nu ska du skriva en emulator för ett enkelt assemblerspråk, som vi kallar PyAssm.

I språket ska vi ha tillgång till ett antal *register* som vi kan se som variabler i språket. Dessa register är fördefinierade och har namnen "A" till "Z" (stora bokstäver). När man startar ett PyAssm-program körs ska varje register sättas till värdet 0 (heltal noll).

- ...ska varje register **SÄTTAS till värdet 0**... *för att det blir lättare då!*

- **Som vi föreslår:**

- `alph = ["A", "B", ..., "Z"]`
- `vars = {letter: 0 for letter in alph}`
 - Eller:
`vars = dict()`
for letter **in** alphabet:
 `vars[letter] = 0`
- För att **hämta ett värde**:
 - *Finns* garanterat i vars
 - `the_value = vars[letter]`

- **Alternativ lösning:**

- `vars = dict()`
 # Initialisering klar!
- För att **hämta ett värde**:
 - **if** letter **not in** vars:
 `the_value = 0`
 else:
 `the_value = vars[letter]`
 - På **många** ställen, lätt att glömma!

Förståelse: Vad sägs om typer?

- Första elementet i en nod är dess **nodvärde**, som kan vara ett godtyckligt Python-värde av godtycklig typ.

■ ...godtyckligt Python-värde...

- Kan alltså vara en lista, ett flyttal, ...
- **Även** om alla exempel bara använder strängar och heltal!



Exempel:

- `assert treeval2(["*", [3, [5, [7]]], [11]]) == 165 # (3+5+7)*11`
- `assert treeval2(["***", [3, [5, [7]]], [11]]) == 1155 # 3*5*7*11`
- `assert treeval2([10, ["*", [5, [6]], [7, [8]]], [15, [10]]]) == 200`
- `assert treeval2([10, ["***", [5, [6]], [7, [8]]], [15, [10]]]) == 1715`

Förståelse: Läs den löpande texten



Vi kan lätt ta reda på om alla element i en sekvens `subseq` också finns i samma ordning i en längre sekvens `seq`, där ytterligare element kan vara infogade mellan dem.

Din första uppgift är att implementera funktionen `find_subseq(subseq, seq)`, som ska returnera en lista av *index* där elementen från den godtyckliga sekvensen `subseq` kan hittas i den godtyckliga sekvensen `seq`. Om detta inte går, ska `None` returneras.

- Vissa returnerade sekvenser när ***element i subseq saknades i seq***
 - ”om alla element i `subseq` också finns...”
- Vissa returnerade sekvenser när ***element inte fanns i rätt ordning***
 - ”...också finns i samma ordning...”

Att testa en lösning

- Två viktiga sätt att skapa korrekta lösningar:
 - **Läs texten**, förstå den, använd din egen förståelse när du **läser** din kod
 - Skapa **testfall**, använd dem när du **kör** din kod

- Tentans testfall är oftast assertions
 - Ska kunna köras *exakt som de står*
 - **Finns redan med i filmallen (ex?.py)**
 - `assert split_fib("abc def ghi j klm nopq rst uv wxy z 123 456 /+") == [['abc'], ['def'], ['ghi', 'j'], ['klm', 'nopq', 'rst'], ['uv', 'wxy', 'z', '123', '456'], ['/+']]`
 - **Skriv inte ut** svaret i din funktion; returnera det!
 - **Förstå uttrycken**
 - `assert nth_sums([1,2,3,4,5]) == [1+2+3+4+5, 2+4, 3, 4, 5]`
 - Samma som:
`assert nth_sums([1,2,3,4,5]) == [15, 6, 3, 4, 5]`
 - Inte samma som:
`assert nth_sums([1,2,3,4,5]) == ["1+2+3+4+5", "2+4", "3", "4", "5"]`

- Fullständiga eller partiella testfall kan också finnas i **texten**

Exempel: Trädvärdet för $(15, [(16, []), (17, [(18, [])])])$ är $15 - (16) + (17 - (18))$, där vi på högsta nivån börjar med $-$ och fortsätter med $+$. Delträdet $(17, [(18, [])])$ behandlas som ett eget träd och dess nodvärde är $17 - (18)$, med subtraktion för det första barnet i delträdet.

- **Använd dem!**

- Som de är, eller genom att skapa egna tester

Men inga testfall kan vara fullständiga!

**Tentans testfall är till stor del skapade
för att förklara och förtydliga vad vi menar i löpande text,
inte för att fånga alla fel ni kan göra**

**Det går att klara alla tentans testfall och få 0 poäng (eller nära 0),
om lösningen har allvarliga fel (men *råkar* få rätt på testfallen)**

- Att skapa egna testfall:

- Behöver inte vara så svårt!

1. Skapa många **indata** som koden kan **köras** med

- Anropa func(...)
- Satsa på **variation** i indata – olika datatyper, olika längder, olika nästlingsdjup, olika stora värden, ... (inom uppgiftens gränser!)
- Går bra även om du inte vet korrekta svaren
- Kanske det **kraschar** – då har du hittat ett fel!

2. När detta fungerar, **titta ”manuellt” på utdata**

- print(func(...))
- Ser det uppenbart fel ut? Då har du kanske hittat ett problem!

3. Sista steget: Testa att det verkligen är **helt korrekta** svar

- Om du kan räkna ut svaren själv

**Går
relativt
snabbt!**

- Testa alltid **specialfall** och **extremfall** (inom uppgiftens gränser)!
 - Tomma listor, tupler, ...
 - Negativa tal... även om exemplen bara har positiva
 - Nästlade listor?
 - Kanske en underlista är tom
 - Kanske många nivåer, få nivåer, ...
 - Långa listor, korta listor
 - Listor i ordning ([1,2,3,4,5]), och osorterat ([1,3,5,2,4])
- Skriv funktionen `sum(seq)` som returnerar summan av alla element i listan `seq...`
 - `Sum([1,2,3]) == 6`
 - `Sum([ [1,2], [3] ]) == [1,2] + [3] == [1,2,3]`

Skapa testfall från uppgiften

Skriv funktionen `split_lists(seq, sizes)`, där:

- Parametern `seq` är en sekvens (lista eller tupel) av $n \geq 1$ godtyckliga element.
- Parametern `sizes` är en *sträng* bestående av siffror ("0" till "9").

■ "Lista eller tupel"?

- OK, bäst att testa med båda!
- Även om tentans testfall bara använder listor...

Parametern `sizes` kan innehålla godtyckliga siffror, även 0, på godtycklig position:

- `assert split_lists([1, 2, 3], "102") == [[1], [], [2, 3]]`

■ "På godtycklig position"?

- OK, bäst att testa med 0 först och sist också!
- Även om det inte finns med i tentans testfall...

Skapa testfall från uppgiften (2)



Implementera funktionen `find_nested(nl, pred, n)`, som:

- Tar tre argument:
 - Den nästlade listan `nl`. Med “nästlad lista” menar vi en godtyckligt lång lista (typ `list`) där varje element är antingen (a) en nästlad lista, eller (b) en icke-lista, här kallat *baselement*. Ett baselement kan alltså vara av vilken typ som helst utom `list`.
- ”Nästlade listor”?
 - OK, bäst att testa med djupa listor (4, 5, 6, ...), även om testfallen bara hade djup 2-3...

Skapa testfall från uppgiften (3)



Din första uppgift är att implementera funktionen `find_subseq(subseq, seq)`, som ska returnera en lista av *index* där elementen från den godtyckliga sekvensen `subseq` kan hittas i den godtyckliga sekvensen `seq`. Om detta inte går, ska `None` returneras.

- ”Godtycklig sekvens”?
 - OK, testa med tupler och listor, och kanske strängar...
 - Och elementen kan alltså vara vad som helst...
 - Fungerar min lösning med heltal, strängar, tupler, ...?

Skapa testfall: Loopar

- Definiera funktionen `fac(n: int)`, som tar ett heltal `n`...
 - `for n in range(100):`
 `print(fac(n))`
 - Testar parametrarna 0 till 99
 - Vi vet inte vad rätt resultat kan vara, men testar i alla fall *körbarhet*
 - Kraschar det?

Testning: Inte bara assertions



- Utöver assertions:
 - Kan som sagt *anropa med många olika indata* och **skriva ut**
 - Till och med automatiskt i vissa fall...
for i in **range**(100):
 # Beroende på vad man vill testa, heltal eller lista eller ...
 print(myfunc1(i)) # Kraschar det?
 print(myfunc2([i]))
 print(myfunc3(list(**range**(i)))) # Lister av olika längd
 ...

Testning: Testa delar



- Många uppgifter kan delas i **delar** som kan **testas separat**
 - Via assertions, print-satser, eller annat
 - Så fastna inte i att bara testa slutresultatet!

- Dela upp i mindre delar, som kan testas / debuggas

- Implementera funktionen `split_fib(s: str)`, som:

Förkortad text!

- Tar som argument en godtyckligt lång textsträng `s`
 - Delar upp strängen i *ord*, vilket definieras som *sekvenser av tecken som inte är mellanslag*
 - Returnerar en *lista av n - 1 listor av ord*, som vi kan kalla ret. Antalet ord i varje lista (och därmed även *antalet listor*) ska bestämmas av ...
- Vanligt problem:
 - Gör "allt på en gång" – **dela strängen i ord** som **genast läggs in i dellistor**
 - Bättre: Skapa först en lista av ord
 - Ett mindre delproblem som är lättare att hålla i huvudet
 - Kan enkelt testas separat: Anropa med strängar; ser svaren korrekta ut?
 - Kan **känna** att du har kommit en bit på vägen!
 - Dela sedan upp listan i dellistor med korrekt längd!

- Får inte ändra i indata?

- Testa:

- ```
seq = [1,2,3]
```

- ```
print(myfunc(seq))
```

- ```
print(seq) # Blev det någon synlig ändring?
```

# Testning – men inte bara testning!

**Tester är bra,  
men jämför också koden  
med vad uppgiften säger!**

# Fallgropar i rekursion

- Occur = förekomma, hända
- Recur = återkomma, hända igen

```
def facit_treeeval(tree):
 if len(tree) == 1:
 val = tree[0]
 elif tree[0] == "*":
 val = facit_treeeval(tree[1])
 for node in tree[2:]:
 val *= facit_treeeval(node)
 else:
 val = tree[0]
 for node in tree[1:]:
 val += facit_treeeval(node)
 return val
```



# Fallgropar i rekursion: Behövs dubbel?

I denna uppgift ska vi beräkna en funktion av ett *träd*, som representeras/lagras så här:

- En **nod** representeras som en lista med minst 1 element.
- Första elementet i en nod är dess **nodvärde**, som kan vara ett godtyckligt Python-värde av godtycklig typ.
- Kvarvarande element i noden (listan) är nodens **barn**, som också är noder.

## ■ Rekursion är bra för träd...

- Flera använde dubbelrekursion, och gjorde misstag...
- Men uppgiften krävde inte det
- Kan vara enklare att *rekursera nedåt* och *iterera åt höger*
  - Om inte uppgiften säger annat!

```
def facit_treeval(tree):
 if len(tree) == 1:
 val = tree[0]
 elif tree[0] == "*":
 val = facit_treeval(tree[1])
 for node in tree[2:]:
 val *= facit_treeval(node)
 else:
 val = tree[0]
 for node in tree[1:]:
 val += facit_treeval(node)
 return val
```

- Se alltid till att du förstår funktionens tänkbara indata
  - Inte ovanligt med misstag vid rekursion, speciellt dubbelrekursion
- Uppgift: `sum_or_product(seq)`
  - Ta in en lista `seq`
  - Om första elementet i `seq` är `""`, multiplicera elementen i resten av `seq`
  - Annars, addera elementen i hela listan
- Början på en rekursiv lösning (saknar basfall för att "listan är slut"):
  - **def** `sum_or_product(seq):`
    - if** `seq[0] == ""`:
      - return** `seq[1] * sum_or_product(seq[2:])` # ignore the `""`, multiply the rest
    - else:**
      - return** `seq[0] + sum_or_product(seq[1:])`
  - **Vad är felet?**

# Rekursion: Förstå dina indata (2)

- **def** sum\_or\_product(seq):  
    **if** seq[0] == "\*":  
        **return** seq[1] \* sum\_or\_product(seq[2:])  
    **else**:  
        **return** seq[0] + sum\_or\_product(seq[1:])

Få in ["\*", 1, 2, 3]  
Rekursera med [2, 3]  
Börjar inte med "\*",  
så vi adderar 2+3...

- **Mer korrekt (men inte färdigt):**

- **def** sum\_or\_product(seq):  
    **if** seq[0] == "\*":  
        **return** seq[1] \* sum\_or\_product(["\*"] + seq[2:])  
    **else**:  
        **return** seq[0] + sum\_or\_product(seq[1:])
- **Anropas med ["\*", 1, 2, 3],  
rekurserar med ["\*", 2, 3] – ett korrekt delproblem!**

Här är  
delproblemet  
specificerat  
på rätt sätt!

# Fallgropar i programmering

- ”Om x är tomma listan ska du returnera [ ]”
  - Vad är felet här?
    - **if not** x:  
    **return** []
  - ”not x” triggar även för 0, tomma tupeln (), och så vidare...
  - Korrekt kod:
    - **if** x == []:  
    **return** []
- Samma sak med ”not x” för att detektera värdet 0, osv
  - Tänk om man skickade in tomma listan!

# Fallgropar: Index (vanligt problem!)



- Vi vill veta **vilket index** vi har kommit till i en **iteration**
  - Problematisk lösning:
    - **for** element **in** seq:  
    **if** test(element):  
        index = seq.index(element)  
        **print**(index)
  - Varför problematiskt?
    - seq = [12, **34**, 56, 78, 56, **34**, 78]
    - Vill få 0, 1, 2, 3, 4, 5, 6 – indexen för de 7 elementen
    - Kan inte skilja på 34 och 34, så skriver ut 0, 1, 2, 3, 2, 1, 3...
  - Korrekta lösningar:
    - **for** index, element **in** enumerate(seq):  
    ...
    - **for** index **in** range(len(seq)):  
    element = seq[index]  
    ...

- Hur testar vi om ett index är **inom listans gränser**?
  - Fel sätt:
    - **try:**  
    `element = seq[index]`  
**except** `IndexError`:  
    # Hantera att vi är utanför gränserna
  - Upptäcker om vi har för stora tal (`index == 5000`)
  - Men vad händer för `seq==[4,5,6]`, `index==-1`?

- Liknande problem:

- Vi vill växla mellan att göra något med seq1 och att göra något *annat* med seq2
- **def** alternate(seq1, seq2):  
    ...  
    **if** (seq == seq1):  
        # We know seq points to the first argument seq1  
    **else**:  
        # We know seq points to the second argument seq2

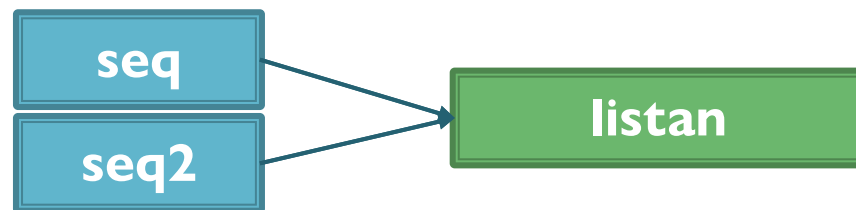
- Problem:

- Om seq1 == seq2 (innehåller lika element) kan vi inte skilja på dem så här
- Kan använda **seq is seq2**
- Kan göra något **helt annat**:  
    Ha en boolesk variable som är sann om det är seq1 vi arbetar med

- Vi vill skapa en kopia av en sekvens

- Problematisks lösning:
  - **def** do\_something(seq):  
    seq2 = seq  
    ...

- Varför problem?
  - Detta skapar ingen kopia!



- **Om** du verkligen behöver en kopia:
  - Använd seq.copy(), copy.copy(seq), ...

Ofta onödigt att kopiera;  
gör det bara om du ändrar inuti ett värde

- Undvik **globala variabler**

- Kan göra att en funktion inte kan köras flera gånger
  - `print(f(12))` → 24 # första gången
  - `print(f(12))` → krasch, eller 42, eller... # andra anropet

Är du osäker?

Testa att anropa funktionen  
flera gånger (samma argument)!

Kör alltid många testfall i rad

- Se också upp med defaultargument, speciellt modifierbara
  - **def** append(element, seq=[]):  
    seq.append(element)  
    return seq
  - **print**(append(5, [1,2,3]))  
    → [1, 2, 3, 5]
  - **print**(append(5))  
    → [5]
  - **print**(append(5))  
    → [5, 5]
  - **print**(append(5, [1,2,3]))  
    → [1, 2, 3, 5]
  - **print**(append(3))  
    → [5, 5, 3]
  - Vad hände? Ska inte default vara [ ]?

Är du osäker?  
Testa att anropa funktionen  
flera gånger!

# Fallgropar: Gapa över för mycket



- Gapa inte över för mycket -- dela upp i mindre delar
  - Diskuterat under testning: `split_fib(s: str)`
    - Dela upp en sträng i ord (separat hjälpfunktion)
    - Lägg lagom många ord i varje dellista
  - Ger mindre delproblem som är lättare att hålla i huvudet!

**Förkortad uppgift!**

En sak till...

# Har funktionen returnerat något?



Implementera funktionen `find_nested(nl, pred, n)`, som:

- Returnerar `(None, None)` om det finns strikt färre än `n` intressanta baselement i `nl`.

## ■ Varför inte bara säga att ni ska returnera **None**?

- **None** kan returneras av många anledningar
  - **return** `nl[5]` # Elementet råkade vara `None`
  - ...eller för att vi *glömmer att returnera något*  
(en funktion som avslutas utan **return** → returnerar i praktiken `None`)
- Värdet `(None, None)` är så "udda" att det sällan returneras "av misstag"
  - Vi kan skilja på om ni *avsiktligt* returnerade `(None, None)`  
eller *råkade* missa att returnera något

**Att plugga med gamla tentor**

## ■ Gamla tentor finns det gott om...

- Ordinarie tenta, 2023-01-10: [Tentamen, Lösningar](#)
- Omtenta (på plats), 2022-08-16: [Tentamen, Lösningar](#)
- Omtenta (på plats), 2022-03-16: [Tentamen, Lösningar](#)
- Ordinarie tenta (på plats), 2022-01-11: [Tentamen, Lösningar, Testfall](#)
- Omtenta (distan), 2021-08-17: [Tentamen, Lösningar, Diskussion](#)
- Omtenta (distan), 2021-03-16: [Tentamen, Lösningar, Diskussion](#)
- Ordinarie tenta (distan), 2021-01-12: [Tentamen, Lösningar, Diskussion](#)
- Omtenta (distan), 2020-08-18: [Tentamen, Lösningar, Lösningssförslag](#)
- Omtenta (på plats), 2020-03-17: [Tentamen, Lösningar, Lösningssförslag](#)
- Ordinarie tenta, 2020-01-17 (14): [Tentamen, Lösningar, Lösningssförslag](#)
- Ordinarie tenta, 2020-01-17 (08): [Tentamen, Lösningar, Lösningssförslag](#)
- Omtenta, 2019-08-20: [Tentamen, Lösningar](#)
- Omtenta, 2019-04-24: [Tentamen, Lösningar](#)
- Ordinarie tenta, 2019-01-17 (14): [Tentamen, Lösningar, Lösningssförslag](#)
- Ordinarie tenta, 2019-01-17 (08): [Tentamen, Lösningar, Lösningssförslag](#)
- Omtenta, 2018-08-22: [Tentamen, solutions.py testfall.txt](#)
- Omtenta, 2018-04-04: [Tentamen, solutions.py](#)
- Ordinarie tenta, 2018-01-12: [Tentamen, solutions.py Rättningsmall](#)
- Omtenta, 2017-08-16: [Tentamen, solutions.py Rättningsmall](#)
- Omtenta, 2017-04-19: [Tentamen, solutions.py graph.py Rättningsmall](#)
- Ordinarie tenta, 2017-01-10, fm: [Tentamen, solutions.py Rättningsmall](#)
- Ordinarie tenta, 2017-01-10, em: [Tentamen, solutions.py Rättningsmall](#)
- Omtenta, 2016-08-17: [Tentamen, solutions.py Rättningsmall](#)
- Omtenta, 2016-03-30: [Tentamen, solutions.py Rättningsmall](#)
- Ordinarie tenta, 2016-01-12, fm: [Tentamen, solutions.py Rättningsmall](#)
- Ordinarie tenta, 2016-01-12, em: [Tentamen, solutions.py](#)
- Omtenta, 2015-04-10: [Tentamen, solutions.py](#)
- Ordinarie tenta, 2015-01-13, em: [Tentamen, graph.py, grammatik\\_s.py, solutions.py](#)
- Ordinarie tenta, 2015-01-13, fm: [Tentamen, graph.py, poly.py, solutions.py](#)
- Omtenta 2014-04-25: [Tentamen, egginator\\_s.py, solutions.py](#)
- Ordinarie tenta, andra passet 2014-01-15: [Tentamen, match\\_s.py, solutions.py](#)
- Ordinarie tenta, första passet 2014-01-14: [Tentamen, discrim\\_s.py, solutions.py](#)
- Omtenta 2013-08-21: [tenta.txt, tree.py, huffman\\_s.py, solutions.py](#)
- Omtenta 2013-03-27: [tenta.txt, poly.py, solutions.py](#)
- Ordinarie tenta 2012-12-18: [tenta.txt, solutions.py](#)
- Exempeltenta: [exempeltenta.pdf, tictactoe\\_s.py, solutions.py](#)

- Användbart för att öva, men kurser förändras...
  - 2012-2018: Annan examinator
  - 2020-2021: Tenta på distans, annorlunda uppgifter
  - Allmänna förändringar och justeringar

- Tidigare tentor: Krav på rekursion som lösningsmodell
  - Måste använda rekursion för att få poäng

## Deluppgift 2a: Rekursiv implementation (2.5p)

Den rekursiva implementationen, `fusc_r(n: int)`, är antagligen den enklaste i denna uppgift: Den matematiska definitionen är ju redan rekursiv, så du behöver helt enkelt överföra den matematiska notationen till en definition av en Python-funktion.

Testfall finns på nästa sida.

## Deluppgift 2b: Iterativ implementation (2.5p)

I den andra deluppgiften skapar du istället en iterativ implementation, `fusc_i(n: int)`. Denna funktion får alltså *inte* anropa sig själv (eller anropa en hjälpfunktion som sedan anropar sig själv), utan ska vara rent iterativ.

- Nu: Använd rekursion när det underlättar
  - Hantera nästlade listor, trädstrukturer, ...
  - Svårt att hantera godtyckligt djup med iteration
  - (Går bra att undvika dubbelrekursion)

I denna uppgift vill vi hantera *listor med samma nästlade struktur*. Med detta menar vi t.ex.:

- `seq1 = [1,2,3],`  
`seq2 = [9,8,7]`
- `seq1 = [[["a"], 6, [2, (3, 5)]]],`  
`seq2 = [[["b"], 5, [1, (1, 1, 42)]]]`

Mer formellt har två listor `seq1` och `seq2` "samma nästlade struktur" om och endast om (1) de är lika långa, och (2) ett av följande villkor gäller för elementen `elem1=seq1[pos]` och `elem2=seq2[pos]` på alla positioner `pos` i de två listorna:

Frågor?