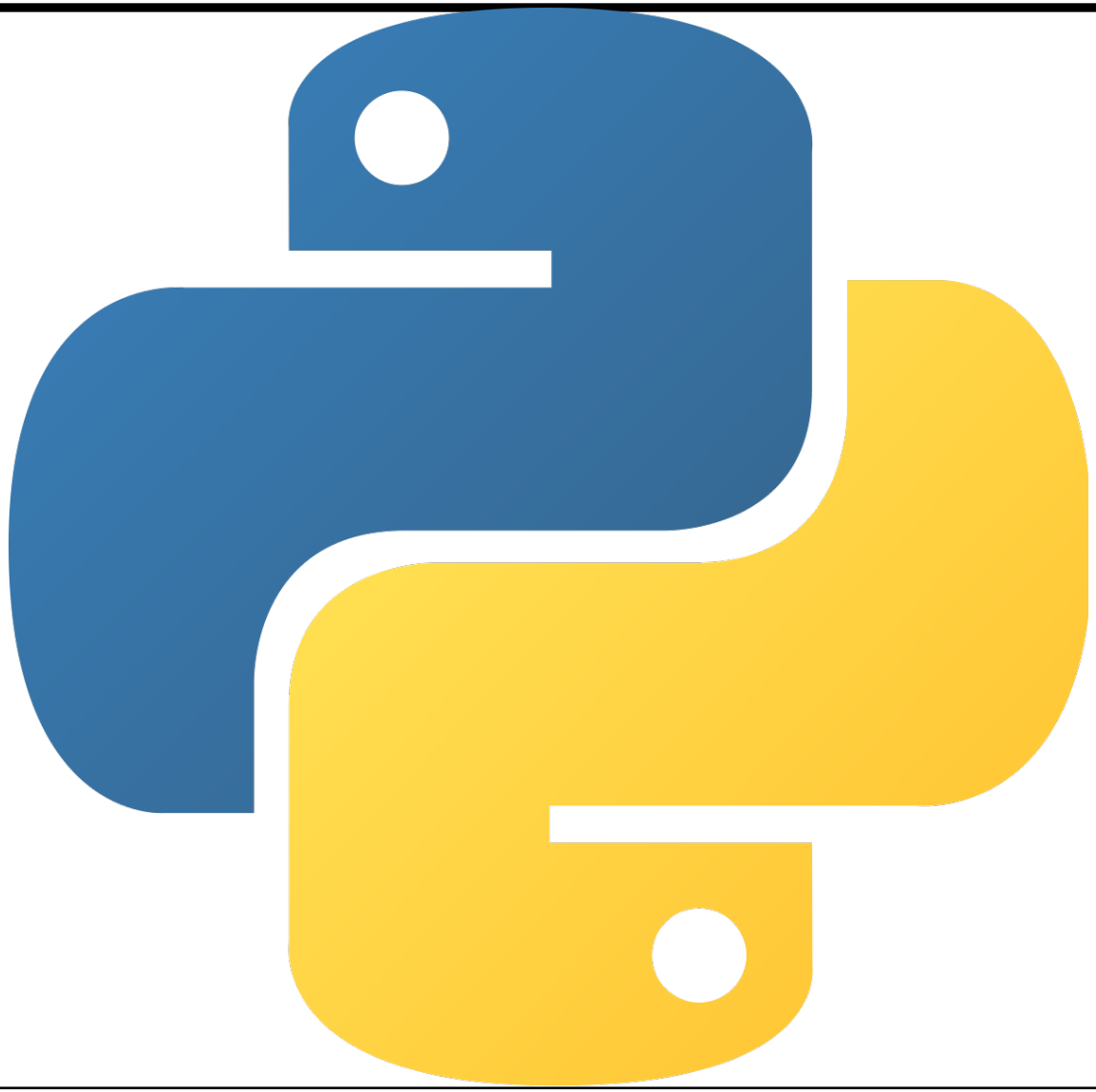
DAGENS FRÅGA

PYTHON PROGRAMMERING



Föreläsning 10

DAGENS AGENDA

- Error och exceptions: skapa egna och hantera
- Debugging
- Att testa kod
- Unit testing
- Pytest

FÖRRA FÖRELÄSNING

ERRORS OCH EXCEPTIONS

- Syntax error: Syntaxfel som gör att koden inte kan köra för att den inte är skriven rätt
- Runtime error: Fel som upptäcks medan kod körs
- Semantisk (logiskt) error: Fel i logiken som gör att programmet inte gör rätt. Dessa kan vara svåra att upptäcka eftersom programmet inte avslutas som med de två andra typerna!

- Vanliga runtime errors:
  - *NameError:* Fel variabelnamn
  - *TypeError:* Fel typ på input
  - *ValueError:* Rätt typ men fel värde
  - *KeyError:* Key finns inte i dictionary
  - *AssertionError:* assert False
  - *ImportError:* Paket finns inte

SKAPA EGNA EXCEPTIONS

- För att skapa en exception kan vi använda nyckelordet **raise**

```
if type(x) != int:
    raise ValueError('The value of x was not an int')
```

- AssertionErrors kommer från nyckelordet **assert** som kan användas för att kolla att ett uttryck stämmer

```
assert cond == True, 'The condition was False'
```

HANTERA EXCEPTIONS

- I python hanterar man exceptions med **try** och **except**-satser
- Strukturen är:

```
try:
    # Denna kod körs alltid

except:
    # Om python stötte på errors under try
    # Så kommer denna kod att köras

finally:
    # Denna kod kommer att köras oavsett vad som hänt innan
```

- I except-satsen får vi en chans att ge användaren feedback eller rätta till felet
- finally-satsen kommer alltid att köras oavsett
- Vi kan även specificera vilket error vi vill att python ska fånga

```
try:
    # kod

except NameError as e:
    # Här fångar vi endast NameError
    # och vi printar den utan att programmet avslutas
    print(e)
```

HANTERA FLERA EXCEPTIONS

- Vi kan också hantera flera olika sorters exceptions genom att göra en kedja av except-block
- Kan användas om vi vet att det kan uppstå en rad olika exceptions

```
try:
    # Denna kod körs alltid

except NameError as e:
    # Här kan vi hantera NameErrors

except AssertionError as e:
    # Här kan vi hantera AssertionError

except TypeError as e:
    # Här kan vi hantera TypeError
```

STACK TRACE

- En stack trace är det meddelande vi får då vi får en error
- I början kan den verka skrämmande och svårtolkad men det är värt att lära sig läsa den!
- Stack tracen läses från botten upp i python – skiljer sig från andra språk
- Den sista raden är det faktiska felet
- Stack tracen listar alla funktionsanrop som gjorts innan felet

```
~/projects/py/Environments/my_env 35s
my_env ❯ python example.py
Traceback (most recent call last):
  File "example.py", line 5, in <module>
    say('Michael')
  File "example.py", line 3, in say
    print('Hello, ' + nam)
NameError: name 'nam' is not defined
```

DEBUGGING

- Debugging är en process där man hittar fel och buggar i ett program
- Det finns många olika sätt att debugga

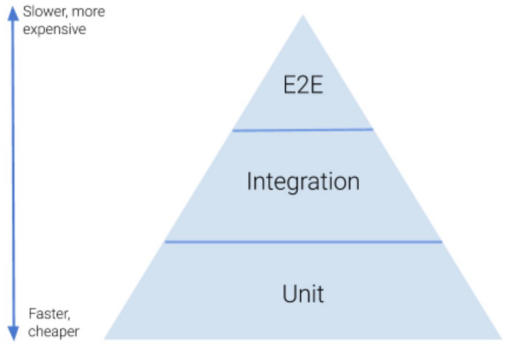
1. Print debugging: använd print() för att kolla värden
    - Lätt men ofta ineffektivt
2. Python debugger – pdb
    - Breakpoints – koden kör fram hit
    - Du kan stega igenom koden själv
    - Du kan köra pythonkod under tiden, t.ex. för att inspektera en variabel
    - Går att använda i jupyter notebooks!
3. Vscode debugger
    - Alla features som pdb har
    - Du kan se all kod i sitt sammanhang
    - Variabler direkt i en lista

ATT TESTA KOD MANUELLT

- När man utvecklar kodbas och lägger till funktionalitet är det viktigt att säkerställa att inga nya buggar tillförts
- Testa manuellt: stoppa in data i ett system/funktion och verifiera att resultatet är rimligt
- Detta kan fungera, men är det effektivt? Oftast inte!

- Att testa kod manuellt fungerar men är sällan effektivt när man utvecklar en applikation

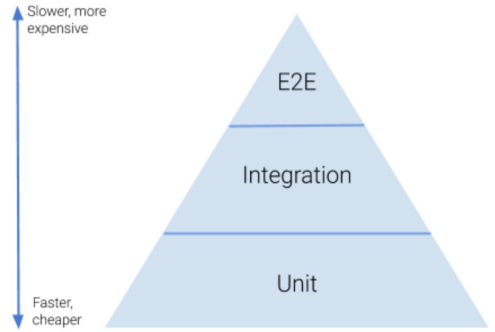
ATT TESTA KOD AUTOMATISKT



- För att testa kod automatiskt måste vi skriva tester som kan köras – alltså mer kod
- Fördelar:
  - Många tester kan köras snabbt och effektivt
  - Vi kan köra tester automatiskt
- Nackdelar:
  - Vi måste definiera tester i kod
  - Dåliga tester (eller buggar) kan ge ett falskt sken av att systemet funkar

UNIT TESTING

- Unit testing är den enklaste och mest använda sortens testning inom mjukvaruutveckling
- Som namnet antyder handlar det om att testa små enheter av koden – enstaka funktioner eller moduler – istället för att testa ett helt system
- För att kunna göra unit tests måste koden vara modulär
  - *Anledningen är att vi vill isolera all funktionalitet*
- Med bra unit testing kan vi automatiskt testa units varje gång vi gör en commit till koden och se om alla komponenter fungerar som det ska



PYTEST

- Pytest är ett välanvänt bibliotek för att göra unit testing i python
- Lätt att använda- för att göra unit tests med pytest skriver vi bara vanliga pythonfunktioner
  - Det lättaste sättet är att använda *assertions*
- Pytest letar automatiskt upp testfunktioner som följer ett standardformat:
  - Filer: *test_*.py* eller **_test.py*
  - Funktioner: vars namn börjar med *test*
- För att gruppera tester kan man även skriva testklasser
- Då man kör pytest kommer det att rapportera alla tester som inte passerar

PYTEST OUTPUT