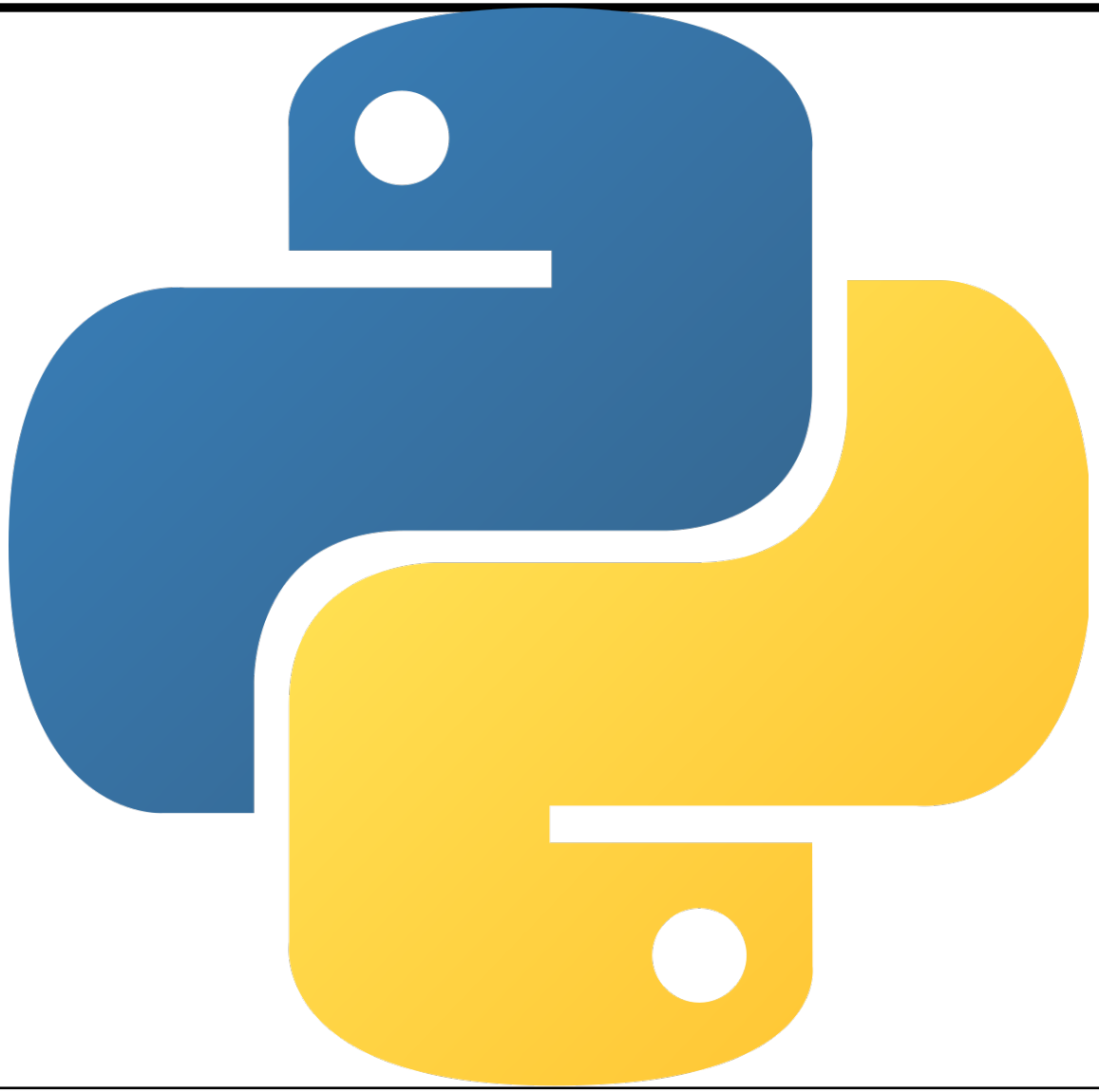

Vad ska ni i julen?
DAGENS
FRÅGA

PYTHON PROGRAMMERING



Föreläsning 2

DAGENS AGENDA

- List
- Tuples
- Dictionaries
- Print and read from console
- Conditions if else
- For and while loops
- Out of range error
- Main
- Functions
- Errors and exception

FÖRRA FÖRELÄSNING

- Introduktion kursen: studieplanering och kursmaterial
- Bekantade med datorens terminal: Powershell
- Installerade Python med Anaconda (minicona)
- Aktiverade Virtual Environments
- Installerade Visual Studio Codes (VSC)
- Aktiverade Python i VSC

LIST

- The list is the most versatile datatype available in python and is often the default choice when selecting a data structure
- Python lists are ordered and mutable. It allows duplicate members
- The elements in a list does not have to be of the same type.
- Elements in a list can be accessed using indices
- Indices values are determined by the offset from the starting position so for example index 0 is the first position of the array
- To add an item to the end of a list use the *.append()* function
- Lists can be printed directly in which case the lists is printed with square brackets with commas separating the elements in the list

```
inputs = [0 , "1", 2]
print(inputs) # prints [0, '1', 2]
inputs.append(3)
inputs[1] = 1
# changes the value of index 1 ("1")
print(inputs) # prints [0, 1, 2, 3]
```

MORE ON LIST

- To get the length of a sequence use the *len()* function
- To concatenate two sequences together you can use the + operator
- To check if a value is in a sequence you can use the keyword *in*
- The *remove()* function removes the first instance of the given value or object
- The *del* keyword can be used to remove an element at a specific index
- To print a sequence without brackets use the *str.join(list)* function which will insert the *str* between each element in a sequence

```
def list_extended():
    inputs = [0, 1]
    print(len(inputs)) # prints 2
    inputs = inputs + [2, 3]
    print(inputs) # prints [0, 1, 2, 3]
    if 0 in inputs:
        print("0 is in inputs") # will be printed
    inputs.remove(0) # removes the first 0 in list
    print(inputs) # prints [1, 2, 3]
    del inputs[1] # del index 1 (2)
    print(inputs) # prints [1, 3]
    print(', '.join(inputs)) # prints 1, 3
```

TUPLES

- Tuples are a collection of elements that is ordered indexed but can not be modified once created(immutable)
- The common sequence functions works for tuples
- Tuples are denoted with parentheses
- Because tuples cannot be modified any operation performed on them that would change them actually returns a new tuple that needs to be assigned to a variable
- Tuples can be used in for loops and supports the *in* keyword
- When returning multiple values a tuple is returned containing them

```
tup1 = (12, 34.56)
tup2 = ('abc', 'xyz')

# not valid for tuples changes values
# tup1[0] = 100

print('abc' in tup2) #prints True

# assigning result to new tuple
tup3 = tup1 + tup2
print(tup3) # prints 12, 34.56, 'abc', 'xyz'
```

DICTIONARIES

- Dictionaries are a collection of key-value pairs. They are very efficient when you want to associate some data with a key. A key can be anything but must be unique.
- Dictionaries are not sequences.
- Dictionaries are ordered and mutable. It does not allow duplicate keys but allows duplicate values.
- Keys in dictionaries can be of any immutable type most built-in types are immutable. Key does not have to be of the same type
- Values in a dictionary does not have to be of the same type
- When accessing dictionaries you use keys and square brackets. Elements in dictionaries cannot be accessed by index
- To add a key-value pair to an element the key and value must be set at the same time.

```
car = {} # empty dict
car = {
    "brand": "Lancia",
    "model": "Delta S4"
} # dict with start values
print(car["brand"]) # prints Lancia
car["year"] = 1985
print(car)
#prints {'brand': 'Lancia', 'model': 'Delta S4', 'year': 1985}
```

MORE ON DICTIONARIES

- *Len()* returns the number of keys in a dictionary
- Using the *in* keyword will check if the key is in the dictionary
- The *del* keyword can be used to remove a key-value pair from a dict
- To access a key-value pair you can also use the *get(key, default)* function that will try to access the value of a key, if the key does not exist the default value will be returned instead

```
car = {
    "brand": "Lancia",
    "model": "Delta S4"
}
print(len(car)) # prints 2
if "brand" in car:
    print(car["brand"]) # prints Lancia, will be printed
del car["brand"] # deletes the key-value pair "brand"
brand = car.get("brand", "No brand set")
# brand does not exists will return default "No brand set"
print(brand) #prints No brand set
```

PRINT TO CONSOLE

- *print()* can be used to write to the console.
- When printing you can add variables that should be printed with a text by adding them as parameters.
- Use f-strings

```
print(f"Strings {var_name} more strings")
```

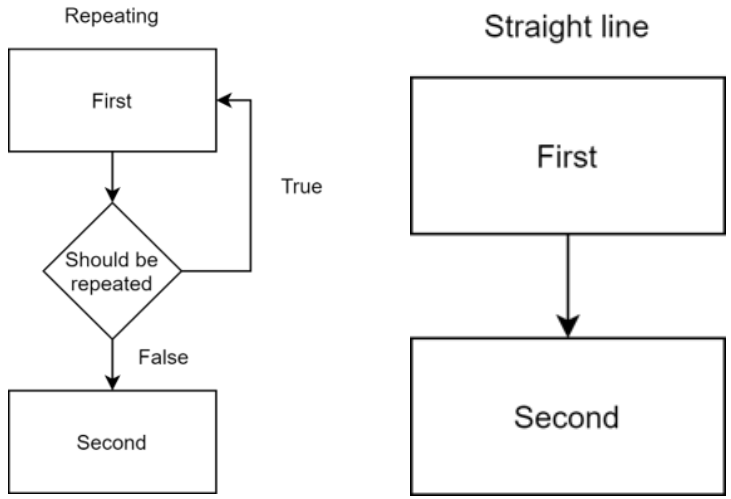
READ FROM CONSOLE

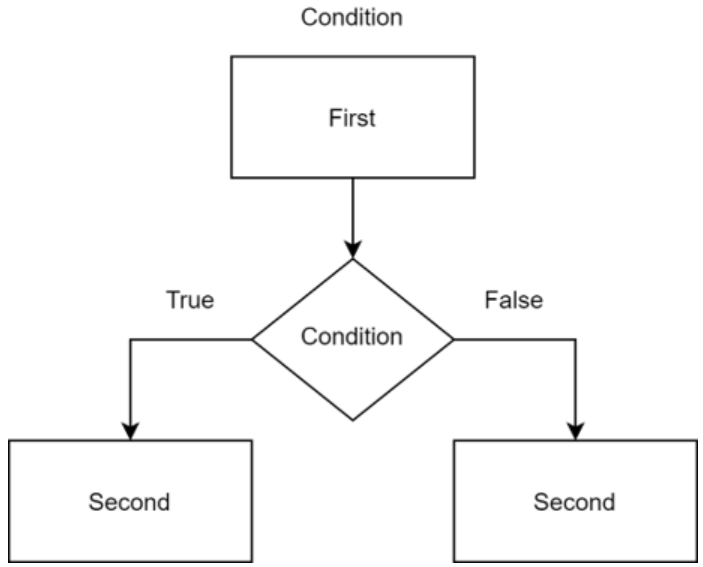
- *input()* can be used to read input from a user using a console
- In the *input()* function you can place a string that the user will see when this line is executed. Use it to so that the user know what they should enter
- When its finished it will return a string that must be passed to a variable

```
user_age = input("Enter your age: ")
user_name = input("Enter your name: ")
print("User inputs:", user_age, user_name, sep=' &')
print("Age: {}, Name: {}".format(user_age, user_name))
```

PROGRAM FLOW

- Up to this point all examples have been very simple they have run straight through from the first statement to the last and the stopped.
- Now we will introduce ways to control the flow of the program. There are three types of program flow.
  - Straight line(sequential)
  - Chosen depending on a condition
  - Repeated according to a give condition





CONDITION

- A condition is something that can be evaluated to either a Truthy or Falsy value. Values that are evaluated to **False** are considered Falsy, values that are evaluated to **True** are considered Truthy. Besides this Python also includes some extra things under Truthy and Falsy.
- When a condition is satisfied the value that is returned is **True** if not the value that is returned is **False**
- To achieve this we can for example use relational operators such as equals, less than and greater than to create conditions that result in **True** or **False**.

CONDITIONAL EXECUTION
IF-ELSE STATEMENT

- If-else statements consists of two blocks: an **if** block that will be executed if the condition is true and an **else** block that will be executed if the condition is false
- A condition is something that can be evaluated to either **True** or **False**. The simplest condition is therefore just the keyword **True** or **False**. This is valid but rather useless as the result will always be the same
- If statements does not have to be followed by an else statement
- Code execution continues normally after the if-else block regardless of which block was executed

```
if True:
    print("Condition is True")
    # Will be printed
else:
    print("Condition is false")
    # Will not be printed
print("If block completed")
# Will be printed
```

RELATIONAL OPERATORS

- **Not equal !=** Will return true if the condition on the left side is not equal to the condition on the right side
- **Less than <** Will return true if the condition on the left side is less than the condition on the right side
- **Less than or equal <=** Works the same as **Less than** but will also return true if they are equal
- **Greater than >** Will return true if the condition on the left side is greater than the condition on the right side
- **Greater than or equal >=** Works the same as **Greater than** but will also return true if they are equal

```
if 1 != 2:
    print("Not equal")
    # Will be printed
if 1 < 2:
    print("Less than")
    # Will be printed
if 1 <= 1:
    print("Less than or equal")
    # Will be printed
if 2 > 1:
    print("Greater than")
    # Will be printed
if 2 >= 2:
    print("Greater than or equal")
    # Will be printed
```

CONDITIONAL LOGICAL OPERATORS

- **AND** *and* Will return True if the condition on the right side is True AND the condition on the left side is True. If the condition on the left side is not True, the right side is not evaluated
- **OR** *or* Will return True if the condition on the right side is True OR the condition on the left side is True. If the condition on the left side is True, then the right side is not evalutead
- **NOT** *not* Will reverse the result so if a condition where to return True it would instead return False

```
if True and True:
    print("True and True == True")
    # Will be printed
if True or False:
    print("True or False == True")
    # Will be printed
if False or True:
    print("False or True == True")
    # Will be printed
if not False:
    print("not False == True")
    # Will be printed
```

IF-ELIF-ELSE STATEMENT

- **Else if** *elif* blocks can be added to check for more conditions, they are evaluated after the if statement from top to bottom so if more than one condition is true only the topmost block will be executed.
- There is no limit to how many *elif* block can be used. But remember the larger an if statement if the harder it is to reason about
- If all *if* and *elif* blocks are false, the else block will be executed

```
if False:
    print("If statement")
elif False:
    print("First elif statement")
elif True:
    print("Second elif statement")
    # Will be printed
else:
    print("Else statement")
```

ITERATION STATEMENTS, LOOPS

- When you need to repeat something several times for example reading input until the user is finished. You would use an iteration statement also known as a loop
- Python have two different loop types: for loops and while loops
- They work a bit differently but essentially does the same thing I.e iterate over a code block until an end condition is meet or the loop is aborted
- If loops end condition are not meet, they will not stop. This is called an infinite loop or endless loop. A hanged program is a program that is stuck in an infinite loop. To abort a hanged python script use ctrl+c

FOR-LOOPS

- For-loops in python are quite different from other programming languages implementation of for loops
- For-loops in python are used to iterate over a sequence for example a list, a string, a set or a range
- A temporary variable is created which holds the value of the current position in the sequence (***i*** in the examples)
- The range() function is used to iterate a specified number of times it returns a sequence of number.
- Its default behaviour is starting at 0 and incrementing by 1 the input of range() is the max numbers it should be generated to, not including it. Using range(3) would return 0, 1, 2

```
hello = "Hello"
for char in hello:
    print(char)
# Prints
# H
# e
# l
# l
# 0

numbers = [0, 1 ,2]

for i in numbers:
    print(i)
for i in range(3):
    print(i)
# Both of these will print
# 0
# 1
# 2
```

FOR-LOOPS AND DICTIONARIES

- Dictionaries can also be iterated over using for loops. Using the same syntax as for lists would result in iterating over the keys in the dictionary
- To iterate over the values you can use the *values()* function
- To iterate over both key and value you can use the *items()* function when doing this you have to use two temporary variables

```
car = {
    "brand": "Lancia",
    "model": "Delta S4"
}
for key in car:
    print(key, car[key]) # prints the key and value

for value in car.values():
    print(value) # prints the value

for key, value in car.items():
    print(key, value) # prints the key and value
```

WHILE-LOOPS

- While loops are used to execute a block as long as the given condition is true. While loops should be used when you do not know how many times something needs to be repeated
- While loops only consist of a loop condition that must be true for the loop to continue. Before the loop is entered this condition is checked.
- Because while loops only have a condition to stop them they often cause infinite loops. Some of the most common reasons are: Not increasing a variables value. Using equals when checking a number instead of using less than or equal. Setting a condition that cannot be meet.

```
i = 0
# print i while i is less than 3
while i < 3:
    print(i)
    i += 1
# prints
# 0
# 1
# 2
```

OUT OF RANGE ERROR AND KEY ERROR

- When using data structures it is possible to try to access something that does not exist
- When using sequences these errors happen when trying to access an index that is **out of range** I.e trying to read an index that is more than the current length of the sequence
- When using dictionary and trying to access a key that does not exist it is called a **KeyError**
- When you try to do these things python will evaulate the result before reading from memory(range-checking) and throw an exception if the index is out of range or the key does not exist. These errors will be visible in the terminal
- In non-range checking languages reading out of range can be used to read and write to memory if an index was not checked before reading.

```
inputs = [0, 1]
print(inputs[2]) # raises IndexError: list index out of range

car = {
    "brand": "Lancia"
}
print(car["model"]) # raises KeyError : model
```

```
Traceback (most recent call last):
  File "c:/workspace/education/python_HT20/data_structur
es/data_structures.py", line 75, in <module>
    out_of_range_list_error()
  File "c:/workspace/education/python_HT20/data_structur
es/data_structures.py", line 56, in out_of_range_list_er
ror
    print(inputs[2]) # raises IndexError: list index out
 of range
IndexError: list index out of range
```

HANDLING OUT OR RANGE AND KEY ERRORS

- Out of range errors are mostly handled by bounds-checking i.e. making sure the index that you are trying to access is within bounds

- The max index of a sequence can be retreived using *len()*. In Python we can write inequality expressions with relations on both sides of the variable. Which is perfect for bounds-checking.

- Key errors can be handled by using the *get(key, default)* function to access the value. BUT this only applies if there exists a default value that makes sense for that key. If no good default value exist allow the error to happen and figure out why the key does not exist when you need it.

```
car = {
    "brand": "Lancia"
}
model = car.get("model", "No model set")
print(model) # prints "No model set"
```

```
inputs = [0, 1]
index = 3
if 0 <= index < len(inputs): # if 0 <= index and index < len(inputs)
    print("Index in range")
else:
    print("Index out of range") # will be printed
```

PYTHON MAIN

- Many programming languages have a special main functions that acts as the entry point when running the program. On the other hand the python interpreter executes scripts starting at the top of the file.
- But having a defined starting point for the program makes it easier to understand how a program works
- A common code pattern has been defined to faciltate this. Where you check the value of the special __name__ variable which changes based on how the script was executed.
- Using the *if __name__ == "__main__"* idiom means that the main function is only called when the script is excuted directly. So if your script was imported the main function would not run.
- You should be using main functions for all the python scripts you write

```python
def main():
    print("Hello main!") # will be printed

if __name__ == "__main__":
    main() # call the main function
```

FUNCTIONS

- A function is a block of code which only runs when it is called.
- You have already been using functions for example print() and len() which are built-in functions
- In python functions are defined using the *def* keyword have parenthesis and ends with a colon the functions code block must be indented
- You can pass data, known as parameters (or arguments), into a function. There is no limit to how many arguments can be passed but it is recommended to use as few as possible
- Functions name follow the same naming conventions as variables. Their name should describe what they do

```
def say_hello(name):
    print("Hello {}!".format(name))

say_hello("World") # prints Hello World
say_hello("Foo") # prints Hello Foo
```

MORE ON FUNCTIONS

- Function can return data as a result often called an out parameter. To return data use the *return* keyword
- When using multiple parameters the order of the parameters are important. You can also use keyword arguments to make it clearer when using multiple arguments the order of these does not matter
- You can also use a default parameter value which will be used if the parameters is not passed. To set a default parameter use a = they must be the last parameters defined
- Function are one of the most powerful tools in a programmer's arsenal they provide better modularity and a high degree of code reusing. Reducing bugs, making code easier to maintain and reason about
- Functions should do only one thing

```
def combine_name(fname, sname, mname = ""):
    return "{} {} {}".format(fname, mname, sname)

full_name = combine_name("Tim", "Nielsen", "Daldorph")
print(full_name) # prints Tim Daldorph Nielsen
full_name = combine_name(sname="Nielsen", fname="Tim")
print(full_name) # prints Tim Nielsen
```

ERRORS AND EXCEPTION

- When a python program encounters an error during runtime it will raise an exception which will if not caught terminate the program
- You can use a *try/except* block to catch and handle errors. The *as* keyword in except will save the caught error to that variable.
- When handling errors you have to specify which exception should be caught. Because all exceptions are derived from the BaseException class you can use Exception to catch all errors. This is not recommended.
- You can use multiple *except* blocks to catch more errors.
- Error handling is one of the absolutley hardest parts of programming as it requires knowledge of what might happen and knowing what to do when that errors happens.

```python
# Will read input until a number is entered
def read_number_from_user():
    valid_number = False
    while not valid_number:
        try:
            number = int(input("Enter number:"))
            valid_number = True
        except ValueError as e:
            print("Can't convert. Message {}".format(e))
        except Exception as e:
            print("Unkown error. Message {}".format(e))
    return number

number = read_number_from_user()
```

MORE ON EXCEPTION HANDLING

- You can also manually raise exception which can for example be used when your expecting a set of predefined inputs and does not get it
- Together with the *try* block you can also use a *finally* block. A *finally* block will **always** be executed even if no error happens or the error was handled or the program crashes. It will **always** be executed
- *Finally* blocks is often used to clean up external things (such as files or network connections), regardless of whether the use of the resource was successful

```
def divide(x, y):
    try:
        result = x / y
        print("result is", result)
    except ZeroDivisionError:
        print("division by zero!")
    finally:
        print("executing finally clause")

divide(2, 1)
# prints result is 2.0
# executing finally clause
divide(2, 0)
# division by zero!
# executing finally clause
```