Business Intelligence-relaterade programspråk

What is Python?

- It is used for:
  - web development (server-side),
  - software development,
  - mathematics,
  - system scripting.
- What can Python do?
  - Python can be used on a server to create web applications.
  - Python can be used alongside software to create workflows.
  - Python can connect to database systems. It can also read and modify files.
  - Python can be used to handle big data and perform complex mathematics.
  - Python can be used for rapid prototyping, or for production-ready software development.

What is Python?

- Why Python?
  - Python works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc).
  - Python has a simple syntax similar to the English language.
  - Python has syntax that allows developers to write programs with fewer lines than some other programming languages.
  - Python runs on an interpreter system, meaning that code can be executed as soon as it is written. This means that prototyping can be very quick.
  - Python can be treated in a procedural way, an object-oriented way or a functional way.

What is Python?

- Good to know
  - The most recent major version of Python is Python 3. However, Python 2, although not being updated with anything other than security updates, is still quite popular.
  - In this tutorial Python will be written in a text editor. It is possible to write Python in an Integrated Development Environment, such as Visual Studio Code, Thonny, Pycharm, Netbeans or Eclipse which are particularly useful when managing larger collections of Python files.
- Python Syntax compared to other programming languages
  - Python was designed for readability, and has some similarities to the English language with influence from mathematics.
  - Python uses new lines to complete a command, as opposed to other programming languages which often use semicolons or parentheses.
  - Python relies on indentation, using whitespace, to define scope; such as the scope of loops, functions and classes. Other programming languages often use curly-brackets for this purpose.

Köra Python-filer

- Detta gjorde vi senast. Vi skriver Python kod i en fil, t ex `hello.py`, och kör den genom att trycka på Run-knappen.
  - Vi kan även köra vårt program genom att skriva `python hello.py`.

```
C:\Users\Your Name>python helloworld.py
Hello, World!
```

Köra Python-skalet

- Om vi bara skriver `python` kommer vi att hamna i Python-skalet. Där kan vi skriva Python-kod utan filer.
- Avsluta skalet genom att skriva `exit()` eller trycka *ctrl* + *d*.

```
C:\Users\Your Name>python
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello, World!")
Hello, World!
```

Python Syntax

Python Indentation

- Indentation refers to the spaces at the beginning of a code line.
- Where in other programming languages the indentation in code is for readability only, the indentation in Python is very important.
- Python uses indentation to indicate a block of code.

```
if 5 > 2:
  print("Five is greater than two!")
```

- Python will give you an error if you skip the indentation.
- The number of spaces is up to you as a programmer, but it has to be at least one.

Python Variables

- In Python, variables are created when you assign a value to it

```
x = 5
y = "Hello, World!"
```

- Python has no command for declaring a variable.

- Variables do not need to be declared with any particular type, and can even change type after they have been set.

```
x = 4       # x is of type int
x = "Sally" # x is now of type str
```

Python Variables

Data types

- You can get the data type of a variable with the `type()` function.

```
x = 5
y = "John"
print(type(x)) # <class 'int'>
print(type(y)) # <class 'str'>
```

Python Variables

Casting

- If you want to specify the data type of a variable, this can be done with casting.

```
x = str(3)    # x will be '3'
y = int(3)    # y will be 3
z = float(3)  # z will be 3.0
```

Python

Single or Double Quotes

- String variables can be declared either by using single or double quotes.

```
x = "John"
# is the same as
x = 'John'
```

Python
Case-Sensitive

- Variable names are case-sensitive.

```
a = 4
A = "Sally"
# A will not overwrite a
```

Python
Case-Sensitive

- Variable names are case-sensitive.

```
a = 4
A = "Sally"
# A will not overwrite a
```

Python
Variable Names

- A variable can have a short name (like `x` and `y`) or a more descriptive name (`age`, `carname`, `total_volume`). Rules for Python variables:
  - Must start with a letter or the underscore character
  - Cannot start with a number
  - Can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
  - Case-sensitive (age, Age and AGE are three different variables)

- Legal variable names:

```
myvar = "John"
my_var = "John"
_my_var = "John"
myVar = "John"
MYVAR = "John"
myvar2 = ”John"
```

- Illegal variable names:

```
2myvar = "John"
my-var = "John"
my var = "John"
```

Python

Multi Words Variable Names

- Variable names with more than one word can be difficult to read.
- There are several techniques you can use to make them more readable:
- **Camel Case**
  - Each word, except the first, starts with a capital letter.

```
myVariableName = "John"
```

- **Pascal Case**
  - Each word starts with a capital letter.

```
MyVariableName = "John"
```

- **Snake Case**
  - Each word is separated by an underscore character.

```
my_variable_name = ”John"
```

Python
Assign values

- Many Values to Multiple Variables
  - Python allows you to assign values to multiple variables in one line.

```
x, y, z = "Orange", "Banana", "Cherry"
```

- One Value to Multiple Variables
  - And you can assign the same value to multiple variables in one line:

```
x = y = z = "Orange"
```

- Unpack a Collection
  - If you have a collection of values in a list, tuple etc. Python allows you extract the values into variables. This is called unpacking.
  - More on collections later

```
fruits = ["apple", "banana", "cherry"]
x, y, z = fruits
```

Python

Output Variables

- The Python print statement is often used to output variables.
- To combine both text and a variable, Python uses the + character.

```
x = "awesome"
print("Python is " + x)
```

- You can also use the + character to add a variable to another variable.

```
x = "Python is ”
y = "awesome"
z =  x + y
print(z)
```

- If you try to combine a string and a number, Python will give you an error.

```
x = 5
y = "John"
print(x + y)
```

Python

Variable scope - Global variables

- Variables that are created outside of a function (as in all of the examples above) are known as global variables.
- Global variables can be used by everyone, both inside of functions and outside. (More on functions later.)

```
x = "awesome"

def myfunc():
  print("Python is " + x)

myfunc()
```

Python

Variable scope - Local variables

- If you create a variable with the same name inside a function, this variable will be local, and can only be used inside the function. The global variable with the same name will remain as it was, global and with the original value.

```
x = "awesome"

def myfunc():
  x = "fantastic"
  print("Python is " + x)  # Python is fantastic

myfunc()

print("Python is " + x)    # Python is awesome
```

Python

Variable scope - The global Keyword

- Normally, when you create a variable inside a function, that variable is local, and can only be used inside that function.

- To create a global variable inside a function, you can use the global keyword.

```
def myfunc():
  global x
  x = "fantastic"

myfunc()
print("Python is " + x)  # Python is fantastic
```

Python

Variable scope - The global Keyword

- Also, use the global keyword if you want to change a global variable inside a function.

```
x = "awesome"

def myfunc():
  global x
  x = "fantastic"

myfunc()
print("Python is " + x)   # Python is fantastic
```

Python

Setting the Data Type

- In Python, the data type is set when you assign a value to a variable.

Example	Data Type
`x = "Hello World"`	str
`x = 20`	int
`x = 20.5`	float
`x = 1j`	complex
`x = ["apple", "banana", "cherry"]`	list
`x = ("apple", "banana", "cherry")`	tuple
`x = range(6)`	range
`x = {"name" : "John", "age" : 36}`	dict
`x = {"apple", "banana", "cherry"}`	set
`x = frozenset({"apple", "banana", "cherry"})`	frozenset
`x = True`	bool
`x = b"Hello"`	bytes
`x = bytearray(5)`	bytearray
`x = memoryview(bytes(5))`	memoryview

Python

Setting the Data Type

- If you want to specify the data type, you can use the following constructor functions.

Example	Data Type
`x = str("Hello World")`	str
`x = int(20)`	int
`x = float(20.5)`	float
`x = complex(1j)`	complex
`x = list(("apple", "banana", "cherry"))`	list
`x = tuple(("apple", "banana", "cherry"))`	tuple
`x = range(6)`	range
`x = dict(name="John", age=36)`	dict
`x = set(("apple", "banana", "cherry"))`	set
`x = frozenset(("apple", "banana", "cherry"))`	frozenset
`x = bool(5)`	bool
`x = bytes(5)`	bytes
`x = bytearray(5)`	bytearray
`x = memoryview(bytes(5))`	memoryview

Python

Multiline Strings

- You can assign a multiline string to a variable by using three quotes.

```
a = """Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua."""
```

Python

Strings are Arrays

- Like many other popular programming languages, strings in Python are arrays of bytes representing unicode characters.
- However, Python does not have a character data type, a single character is simply a string with a length of 1.
- Square brackets can be used to access elements of the string.
- The first character has the position 0.

```
a = "Hello, World!"
print(a[1])
```

Python

Looping Through a String

- Since strings are arrays, we can loop through the characters in a string, with a `for` loop.

```
for x in "banana":
  print(x)
```

Python

String Length

- To get the length of a string, use the `len()` function.

```
a = "Hello, World!"
print(len(a))
```

Python
Check String

- To check if a certain phrase or character is present in a string, we can use the keyword `in`.

```
txt = "The best things in life are free!"
print("free" in txt)

txt = "The best things in life are free!"
print("expensive" not in txt)
```

Python
Slicing

- You can return a range of characters by using the slice syntax.
- Specify the start index and the end index, separated by a colon, to return a part of the string.

```
b = "Hello, World!"
print(b[2:5])          # llo
```

Python
Slicing

- You can return a range of characters by using the slice syntax.
- Specify the start index and the end index, separated by a colon, to return a part of the string.

```
b = "Hello, World!"
print(b[2:5])          # llo
print(b[:5])           # Hello
print(b[2:])           # llo, World!
```

- Use negative indexes to start the slice from the end of the string.

```
b = "Hello, World!”
print(b[-5:-2])        # orl
```

Python
Modify Strings

```
a = " Hello, World! "

print(a.upper())           #  HELLO, WORLD!
print(a.lower())           #  hello, world!
print(a.capitalize())      #  Hello, world!
print(a.strip())           # Hello, world! (Removes white space.)
print(a.replace("H", ”J")) #  Jello, World!
print(a.split(”,”))        # [' Hello', ' World! ’]
```

Python

Escape character

- You will get an error if you use double quotes inside a string that is surrounded by double quotes.

```
txt = "We are the so-called "Vikings" from the north."
```

- To fix this problem, use the escape character \".

```
txt = "We are the so-called \"Vikings\" from the north."
```

Python
Escape character

- Other escape characters used in Python

Code	Result
\'	Single Quote
\\	Backslash
\n	New Line
\r	Carriage Return
\t	Tab
\b	Backspace
\f	Form Feed
\ooo	Octal value
\xhh	Hex value

Python
Evaluate boolean

```
x = "Hello"
print(bool(x))
```

- Most Values are True
  - Almost any value is evaluated to True if it has some sort of content.
  - Any string is True, except empty strings.
  - Any number is True, except 0.
  - Any list, tuple, set, and dictionary are True, except empty ones.

Python
Evaluate boolean

```
x = "Hello"
print(bool(x))
```

- Some Values are False
  - In fact, there are not many values that evaluate to `False`. Some exceptions:
    - Empty values, such as `()`, `[]`, `{}`, `""`, the number `0`, and the value `None`.
    - And of course the value `False` evaluates to `False`.

Python

Functions can Return a Boolean

```
def myFunction() :
  return True

if myFunction():
  print("YES!")
else:
  print("NO!")
```

Python

Evaluate boolean

- Python also has many built-in functions that return a boolean value, like the `isinstance()` function, which can be used to determine if an object is of a certain data type:

```
x = 200
print(isinstance(x, int))
```

Python

Python Operators

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Identity operators
- Membership operators
- Bitwise operators

Python

Python Arithmetic Operators

Operator	Name	Example
+	Addition	x + y
-	Subtraction	x - y
*	Multiplication	x * y
/	Division	x / y
%	Modulus	x % y
**	Exponentiation	x ** y
//	Floor division	x // y

Python

Python Assignment Operators

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
//=	x //= 3	x = x // 3
**=	x **= 3	x = x ** 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

Python

Python Comparison Operators

Operator	Name	Example
==	Equal	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

Python

Python Logical Operators

Operator	Description	Example
and	Returns True if both statements are true	x < 5 and x < 10
or	Returns True if one of the statements is true	x < 5 or x < 4
not	Reverse the result, returns False if the result is true	not(x < 5 and x < 10)

Python

Python Identity Operators

- Identity operators are used to compare the objects, not if they are equal, but if they are actually the same object, with the same memory location.

Operator	Description	Example
is	Returns True if both variables are the same object	x is y
is not	Returns True if both variables are not the same object	x is not y

Python

Python Membership Operators

- Membership operators are used to test if a sequence is presented in an object.

Operator	Description	Example
in	Returns True if a sequence with the specified value is present in the object	x in y
not in	Returns True if a sequence with the specified value is not present in the object	x not in y

Python

Python Bitwise Operators

- Bitwise operators are used to compare (binary) numbers.

Operator	Name	Description
&	AND	Sets each bit to 1 if both bits are 1
|	OR	Sets each bit to 1 if one of two bits is 1
^	XOR	Sets each bit to 1 if only one of two bits is 1
~	NOT	Inverts all the bits
<<	Zero fill left shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off
>>	Signed right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off

Python

Python Collections (Arrays)

- Arrays are collections of several values.
- There are four collection data types in the Python programming language:
  - **List** is a collection which is ordered and changeable. Allows duplicate members.
  - **Tuple** is a collection which is ordered and unchangeable. Allows duplicate members.
  - **Set** is a collection which is unordered and unindexed. No duplicate members.
  - **Dictionary** is a collection which is unordered and changeable. No duplicate members.

Python
Lists

- Lists are used to store multiple items in a single variable.
- Lists are one of 4 built-in data types in Python used to store collections of data, the other 3 are Tuple, Set, and Dictionary, all with different qualities and usage.
- Lists are created using square brackets.

```
this_list = ["apple", "banana", "cherry"]
print(this_list)
```

Python
Lists

- List Items
  - List items are ordered, changeable, and allow duplicate values.
  - List items are indexed, the first item has index [0], the second item has index [1] etc.
- Ordered
  - When we say that lists are ordered, it means that the items have a defined order, and that order will not change.
  - If you add new items to a list, the new items will be placed at the end of the list.
- Changeable
  - The list is changeable, meaning that we can change, add, and remove items in a list after it has been created.
- Allow Duplicates
  - Since lists are indexed, lists can have items with the same value
- List Length
  - To determine how many items a list has, use the `len()` function

Python

List Items - Data Types

- List items can be of any data type.

```
list1 = ["apple", "banana", "cherry"]
list2 = [1, 5, 7, 9, 3]
list3 = [True, False, False]

list4 = ["abc", 34, True, 40, "male"]
```

Python

Lists - Access Items

```
thislist = ["apple", "banana", "cherry"]
print(thislist[1])   # banana
print(thislist[-1])  # cherry

thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(thislist[2:5]) # ['cherry', 'orange', 'kiwi']
print(thislist[:4])  # ['apple', 'banana', 'cherry', 'orange']

if "apple" in thislist:
  print("Yes, 'apple' is in the fruits list")
```

Python

Lists - Change List Items

```
thislist = ["apple", "banana", "cherry"]
thislist[1] = "blackcurrant"
print(thislist)   # ['apple', 'blackcurrant', ’cherry']
```

Python

Lists - Change List Items

```
thislist = ["apple", "banana", "cherry"]
thislist[1] = "blackcurrant"
print(thislist)   # ['apple', 'blackcurrant', ’cherry']

thislist = ["apple", "banana", "cherry", "orange", "kiwi", "mango"]
thislist[1:3] = ["blackcurrant", "watermelon"]
print(thislist)   # ['apple', 'blackcurrant', 'watermelon', 'orange', 'kiwi',
'mango']
```

Python

Lists - Insert Items

```
thislist = ["apple", "banana", "cherry"]
thislist.insert(2, "watermelon")
print(thislist)  # ['apple', 'banana', 'watermelon', ’cherry']
```

Python

Lists - Add List Items

```
thislist = ["apple", "banana", "cherry"]
thislist.append("orange")
print(thislist)  # ['apple', 'banana', 'cherry', ’orange']
```

Python

Lists - Extend List

- To append elements from another list to the current list, use the `extend()` method.

```
thislist = ["apple", "banana", "cherry"]
tropical = ["mango", "pineapple", "papaya"]
thislist.extend(tropical)
print(thislist)    # ['apple', 'banana', 'cherry', 'mango', 'pineapple', 'papaya']
```

- The extend() method does not have to append lists, you can add any iterable object (tuples, sets, dictionaries etc.).

```
thislist = ["apple", "banana", "cherry"]
thistuple = ("kiwi", "orange")
thislist.extend(thistuple)
print(thislist)   # ['apple', 'banana', 'cherry', 'kiwi', 'orange']
```

Python

Lists - Remove List Items

```
thislist = ["apple", "banana", "cherry"]
thislist.remove("banana")
print(thislist)   # ['apple', 'cherry']
```

Python

Lists - Remove List Items

```
thislist = ["apple", "banana", "cherry"]
thislist.remove("banana")
print(thislist)   # ['apple', 'cherry']

thislist = ["apple", "banana", "cherry"]
thislist.pop(1)
print(thislist)   # ['apple', 'cherry']

thislist = ["apple", "banana", "cherry"]
thislist.pop()
print(thislist)   # ['apple', 'banana']
```

Python

Lists - Remove List Items

```
thislist = ["apple", "banana", "cherry"]
del thislist[0]
print(thislist)   # ['banana', 'cherry']

thislist = ["apple", "banana", "cherry"]
del thislist
print(thislist)   # NameError: name 'thislist' is not defined

thislist = ["apple", "banana", "cherry"]
thislist.clear()
print(thislist)   # []
```

Python

Lists - Loop through lists

```
thislist =
["apple", "banana", "cherry"]
for x in thislist:
  print(x)

# ################

thislist =
["apple", "banana", "cherry"]
for i in range(len(thislist)):
  print(thislist[i])
```

```
thislist =
["apple", "banana", "cherry"]

i = 0
while i < len(thislist):
  print(thislist[i])
  i = i + 1



# ################

thislist =
["apple", "banana", "cherry"]
[print(x) for x in thislist]
```

Python

Lists - List Comprehension

- List comprehension offers a shorter syntax when you want to create a new list based on the values of an existing list.
- Example: Based on a list of fruits, you want a new list, containing only the fruits with the letter "a" in the name.
- Without list comprehension you will have to write a for statement with a conditional test inside:

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
newlist = []

for x in fruits:
  if "a" in x:
    newlist.append(x)

print(newlist)
```

Python

Lists - List Comprehension

- With list comprehension you can do all that with only one line of code:

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]

newlist = [x for x in fruits if "a" in x]
print(newlist)
```

  - Syntax:

```
newlist = [expression for item in iterable if condition == True]
```

  - The condition is like a filter that only accepts the items that valuate to True.

Python

Lists - List Comprehension

- The condition if x != "apple" will return True for all elements other than "apple", making the new list contain all fruits except "apple".
- The condition is optional and can be omitted

```
newlist = [x for x in fruits]
```

- The iterable can be any iterable object, like a list, tuple, set etc.

```
newlist = [x for x in range(10)]

newlist = [x for x in range(10) if x < 5]
```

Python

Lists - List Comprehension

- The expression is the current item in the iteration, but it is also the outcome, which you can manipulate before it ends up like a list item in the new list.

```
newlist = [x.upper() for x in fruits]
```

- You can set the outcome to whatever you like.

```
newlist = ['hello' for x in fruits]
```

- The expression can also contain conditions, not like a filter, but as a way to manipulate the outcome.

```
newlist = [x if x ≠ "banana" else "orange" for x in fruits]
```

- "Return the item if it is not banana, if it is banana return orange”

Python

Lists - Sort lists

- List objects have a sort() method that will sort the list alphanumerically, ascending, by default.

```
thislist = ["orange", "mango", "kiwi", "pineapple", "banana"]
thislist.sort()
print(thislist)   # ['banana', 'kiwi', 'mango', 'orange', 'pineapple']

thislist = [100, 50, 65, 82, 23]
thislist.sort()
print(thislist)   # [23, 50, 65, 82, 100]

thislist = ["orange", "mango", "kiwi", "pineapple", "banana"]
thislist.sort(reverse = True)
print(thislist)   # ['pineapple', 'orange', 'mango', 'kiwi', ’banana']
```

Python

Lists - Customize Sort Function

- You can also customize your own function by using the keyword argument `key = function`.
- The function will return a number that will be used to sort the list (the lowest number first).
- Example: Sort the list based on how close the number is to 50.

```
def myfunc(n):
  return abs(n - 50)

thislist = [100, 50, 65, 82, 23]
thislist.sort(key = myfunc)
print(thislist)
```

Python

Lists - Case-insensitive sort

```
thislist = ["banana", "Orange", "Kiwi", "cherry"]
thislist.sort(key = str.lower)
print(thislist)   # ['banana', 'cherry', 'Kiwi', 'Orange']
```

Python

Lists - Reverse order

```
thislist = ["banana", "Orange", "Kiwi", "cherry"]
thislist.reverse()
print(thislist)   # ['cherry', 'Kiwi', 'Orange', ’banana']
```

Python

Lists - Copy Lists

- You cannot copy a list simply by typing list2 = list1, because: list2 will only be a reference to list1, and changes made in list1 will automatically also be made in list2.

```
thislist = ["apple", "banana", "cherry"]
mylist = thislist.copy()
print(mylist)

thislist = ["apple", "banana", "cherry"]
mylist = list(thislist)
print(mylist)
```

Python

Lists - Join Lists

- You cannot copy a list simply by typing list2 = list1, because: list2 will only be a reference to list1, and changes made in list1 will automatically also be made in list2.

```
list1 = ["a", "b", "c"]
list2 = [1, 2, 3]

list3 = list1 + list2
print(list3)   # ['a', 'b', 'c', 1, 2, 3]
```

Python

Lists - Join Lists

- You cannot copy a list simply by typing list2 = list1, because: list2 will only be a reference to list1, and changes made in list1 will automatically also be made in list2.

```
list1 = ["a", "b", "c"]
list2 = [1, 2, 3]

list3 = list1 + list2
print(list3)   # ['a', 'b', 'c', 1, 2, 3]

# ##############

list1.extend(list2)
print(list1)   # ['a', 'b', 'c', 1, 2, 3]
```

Python
Tuples

- Tuples are used to store multiple items in a single variable.
- A tuple is a collection which is ordered and unchangeable.

```
thistuple = ("apple", "banana", "cherry")
```

Python
Tuples

- Ordered
  - When we say that tuples are ordered, it means that the items have a defined order, and that order will not change.
- Unchangeable
  - Tuples are unchangeable, meaning that we cannot change, add or remove items after the tuple has been created.
- Allow Duplicates
  - Since tuple are indexed, tuples can have items with the same value.

Python
Tuples

- To determine how many items a tuple has, use the `len()` function.

```
thistuple = ("apple", "banana", "cherry")
print(len(thistuple))
```

- Tuple items can be of any data type.

```
tuple1 = ("apple", "banana", "cherry")
tuple2 = (1, 5, 7, 9, 3)
tuple3 = (True, False, False)

tuple4 = ("abc", 34, True, 40, "male")
```

Python

Tuples - Update Tuples

- Tuples are unchangeable or *immutable*, meaning that you cannot change, add, or remove items once the tuple is created, but there are some workarounds, like converting the tuple into a list to be able to change it.

```
x = ("apple", "banana", "cherry")
y = list(x)
y[1] = "kiwi"
x = tuple(y)

print(x)
```

Python

Tuples - Update Tuples

- Tuples are unchangeable or *immutable*, meaning that you cannot change, add, or remove items once the tuple is created, but there are some workarounds, like converting the tuple into a list to be able to change it.

```
x = ("apple", "banana", "cherry")
y = list(x)
y[1] = "kiwi"
x = tuple(y)
```

- You cannot add items to a tuple.

```
thistuple = ("apple", "banana", "cherry")
thistuple.append("orange") # This will raise an error
```

Python

Tuples - Delete Tuples

```
thistuple = ("apple", "banana", "cherry")
del thistuple
print(thistuple) # this will raise an error because the tuple no longer exists
```

Python

Tuples - Unpacking a Tuple

- When we create a tuple, we normally assign values to it. This is called "packing" a tuple.

```
fruits = ("apple", "banana", ”cherry")
```

- Unpacking.

```
fruits = ("apple", "banana", "cherry")
(green, yellow, red) = fruits
```

Python
Tuples - Unpacking a Tuple

- When we create a tuple, we normally assign values to it. This is called "packing" a tuple.

```
fruits = ("apple", "banana", ”cherry")
```

- Unpacking.

```
fruits = ("apple", "banana", "cherry")
(green, yellow, red) = fruits

fruits = ("apple", "banana", "cherry", "strawberry", "raspberry")
(green, yellow, *red) = fruits

print(green)   # apple
print(yellow)  # banana
print(red)     # ['cherry', 'strawberry', ’raspberry']
```

Python

Tuples - Loop a Tuple

```
thistuple = ("apple", "banana", "cherry")
for x in thistuple:
  print(x)

for i in range(len(thistuple)):
  print(thistuple[i])

i = 0
while i < len(thistuple):
  print(thistuple[i])
  i = i + 1
```

Python

Tuples - Join Tuples

```
tuple1 = ("a", "b" , "c")
tuple2 = (1, 2, 3)

tuple3 = tuple1 + tuple2
print(tuple3)
```

Python

Tuples - Multiply Tuples

```
fruits = ("apple", "banana", "cherry")
mytuple = fruits * 2

print(mytuple)  # ('apple', 'banana', 'cherry', 'apple', 'banana', 'cherry')
```

Python

Sets

- Sets are used to store multiple items in a single variable.
- A set is a collection which is both unordered and unindexed.
- Sets are written with curly brackets.

```
thisset = {"apple", "banana", "cherry"}
```

Python
Sets

- Set Items
  - Set items are unordered, unchangeable, and do not allow duplicate values.
- Unordered
  - Unordered means that the items in a set do not have a defined order.
  - Set items can appear in a different order every time you use them, and cannot be referred to by index or key.
- Unchangeable
  - Sets are unchangeable, meaning that we cannot change the items after the set has been created.
  - Once a set is created, you cannot change its items, but you can add new items.
- Duplicates Not Allowed
  - Sets cannot have two items with the same value.
- Set Items - Data Types
  - Set items can be of any data type

Python

Sets - Access Set Items

- You cannot access items in a set by referring to an index or a key, but you can loop through the set items using a for loop, or ask if a specified value is present in a set, by using the in keyword.

```
thisset = {"apple", "banana", "cherry"}

for x in thisset:
  print(x)
```

```
thisset = {"apple", "banana", "cherry"}
print("banana" in thisset)
```

Python
Sets - Add Set Items

- Once a set is created, you cannot change its items, but you can add new items.

```
thisset = {"apple", "banana", "cherry"}
thisset.add("orange")
print(thisset)   # {'orange', 'apple', 'banana', 'cherry'}
```

- To add items from another set into the current set, use the `update()` method.

```
thisset = {"apple", "banana", "cherry"}
tropical = {"pineapple", "mango", "papaya"}
thisset.update(tropical)
```

```
thisset = {"apple", "banana", "cherry"}
mylist = ["kiwi", "orange"]
thisset.update(mylist)
```

Python

Sets - Remove Set Items

- To remove an item in a set, use the `remove()`, or the `discard()` method.

```
thisset = {"apple", "banana", "cherry"}
thisset.remove("banana")
# If the item to remove does not exist, remove() will raise an error.
thisset.discard("banana")
# If the item to remove does not exist, discard() will NOT raise an error.
```

Python

Sets - Loop Sets

- You can loop through the set items by using a `for` loop.

```
thisset = {"apple", "banana", "cherry"}
for x in thisset:
  print(x)
```

Python

Sets - Join Sets

- There are several ways to join two or more sets in Python.
- You can use the `union()` method that returns a new set containing all items from both sets, or the `update()` method that inserts all the items from one set into another.
- Both `union()` and `update()` will exclude any duplicate items.

```
set1 = {"a", "b" , "c"}
set2 = {1, 2, 3}

set3 = set1.union(set2)
print(set3)

set1.update(set2)
```

Python

Sets - Join Sets

- The `intersection_update()` method will keep only the items that are present in both sets.

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}

x.intersection_update(y)
print(x)   # {'apple'}
```

- The `intersection()` method will return a *new* set, that only contains the items that are present in both sets.

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}
z = x.intersection(y)
print(z)   # {'apple'}
```

Python

Sets - Join Sets

- The `symmetric_difference_update()` method will keep only the elements that are NOT present in both sets.

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}

x.symmetric_difference_update(y)
print(x)   # {'cherry', 'google', 'banana', 'microsoft'}
```

- The `symmetric_difference()` method will return a *new* set, that contains only the elements that are NOT present in both sets.

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}
z = x.symmetric_difference(y)
print(z)   # {'cherry', 'banana', 'microsoft', 'google'}
```

Python
Dictionaries

- Dictionaries are used to store data values in key:value pairs.
- A dictionary is a collection which is ordered*, changeable and does not allow duplicates.
- (As of Python version 3.7, dictionaries are ordered. In Python 3.6 and earlier, dictionaries are unordered.)

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
```

Python

Dictionary Items

- Dictionary items are ordered, changeable, and does not allow duplicates.
- Dictionary items are presented in key:value pairs, and can be referred to by using the key name.

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(thisdict["brand"])
```

Python
Dictionary Items

- Ordered or Unordered?
  - As of Python version 3.7, dictionaries are ordered. In Python 3.6 and earlier, dictionaries are unordered.
  - When we say that dictionaries are ordered, it means that the items have a defined order, and that order will not change.
  - Unordered means that the items does not have a defined order, you cannot refer to an item by using an index.
- Changeable
  - Dictionaries are changeable, meaning that we can change, add or remove items after the dictionary has been created.
- Duplicates Not Allowed
  - Dictionaries cannot have two items with the same key

Python

Dictionary Items - Data Types

- The values in dictionary items can be of any data type.

```
thisdict = {
  "brand": "Ford",
  "electric": False,
  "year": 1964,
  "colors": ["red", "white", "blue"]
}
```

Python

Access Dictionary Items

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
x = thisdict["model"]

x = thisdict.get(”model")
```

Python

Access Dictionary Items - Keys & Values

```
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
x = car.keys()
print(x)   # dict_keys(['brand', 'model', 'year'])

y = car.values()
print(y)   # dict_values(['Ford', 'Mustang', 1964])

z = car.items()
print(z)   # dict_items([('brand', 'Ford'), ('model', 'Mustang'), ('year', 1964)])

if "model" in car:
  print("Yes, 'model' is one of the keys in the car dictionary")
```

Python
Change values

```
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
car["year"] = 2018
```

- Update Dictionary
  - The update() method will update the dictionary with the items from the given argument.
  - The argument must be a dictionary, or an iterable object with key:value pairs.

```
car.update({"year": 2020})
```

Python
Adding Dictionary Items

- Adding an item to the dictionary is done by using a new index key and assigning a value to it.

```
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
car.["color"] = "red"
```

  - The update() method will update the dictionary with the items from a given argument. If the item does not exist, the item will be added.

  - The argument must be a dictionary, or an iterable object with key:value pairs.

```
car.update({"color": "red"})
```

Python

Remove Dictionary Items

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
```

- The pop() method removes the item with the specified key name

```
thisdict.pop("model")
```

- The popitem() method removes the last inserted item.

```
thisdict.popitem()
```

- The del keyword removes the item with the specified key name.

```
del thisdict["model"]
```

- The del keyword removes the item with the specified key name or even the entire dictionary.

```
del thisdict["model"]
del thisdict
```

- The clear() method empties the dictionary

```
thisdict.clear()
```

Python

Loop Through a Dictionary

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}

# Print all key names in the dictionary
for x in thisdict:
  print(x)

for x in thisdict.keys():
  print(x)
```

```
# Print all values in the dictionary
for x in thisdict:
  print(thisdict[x])

for x in thisdict.values():
  print(x)

# Loop through both keys and values
for x, y in thisdict.items():
  print(x, y)
```

Python

Copy a Dictionary

- You cannot copy a dictionary simply by typing dict2 = dict1, because: dict2 will only be a reference to dict1, and changes made in dict1 will automatically also be made in dict2. There are some options, though.

```
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
mydict = car.copy()

mydict = dict(car)
```

Python
Nested Dictionaries

- You cannot copy a dictionary simply by typing dict2 = dict1, because: dict2 will only be a reference to dict1, and changes made in dict1 will automatically also be made in dict2. There are some options, though.

```
myfamily = {
  "child1" : {
    "name" : "Rasmus",
    "year" : 2005
  },
  "child3" : {
    "name" : "Joel",
    "year" : 2011
  }
}

print(myfamily["child1"]["name"])
```

Python

Conditions and if statements

- Do something if a condition is fulfilled.

```
a = 33
if 200 > a:
  print("b is greater than a")
```

- The `elif` keyword is pythons way of saying "if the previous conditions were not true, then try this condition”.

```
a = 33
b = 33
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
```

- The else keyword catches anything which isn't caught by the preceding conditions.

```
a = 200
b = 33
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
else:
  print("a is greater than b”)
```

- Shorthand

```
if a > b: print("a is greater than b")

print("A") if a > b else print("B")

print("A") if a > b else print("=") if a ==
b else print("B")
```

Python

Conditions and if statements

- Nested if - if statements inside if statements

```
if x > 10:
  print("Above ten,")
  if x > 20:
    print("and also above 20!")
  else:
    print("but not above 20.")
```

- if statements cannot be empty, but if you for some reason have an if statement with no content, put in the pass statement to avoid getting an error.

```
if b > a:
  pass
```

Python
While-loops

- With the while loop we can execute a set of statements as long as a condition is true.

```
i = 1
while i < 6:
  print(i)
  i += 1
```

- With the break statement we can stop the loop even if the while condition is true.

```
i = 1
while i < 6:
  print(i)
  if i == 3:
    break
  i += 1
```

- With the continue statement we can stop the current iteration, and continue with the next.

```
i = 0
while i < 6:
  i += 1
  if i == 3:
    continue
  print(i)
```

- With the else statement we can run a block of code once when the condition no longer is true.

```
i = 1
while i < 6:
  print(i)
  i += 1
else:
  print("i is no longer less than 6")
```

Python
For-loops

- A for loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).
- This is less like the for keyword in other programming languages, and works more like an iterator method as found in other object-orientated programming languages.
- With the for loop we can execute a set of statements, once for each item in a list, tuple, set etc.

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  print(x)
```

- Even strings are iterable objects, they contain a sequence of characters.

```
for x in "banana":
  print(x)
```

- With the break statement we can stop the loop before it has looped through all the items.

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  print(x)
  if x == "banana":
    break
```

- With the continue statement we can stop the current iteration of the loop, and continue with the next.

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  if x == "banana":
    continue
  print(x)
```

Python
For-loops

- To loop through a set of code a specified number of times, we can use the `range()` function.

- The `range()` function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.

```
for x in range(6):
  print(x)
```

- Note that `range(6)` is not the values of 0 to 6, but the values 0 to 5.

- The `range()` function defaults to 0 as a starting value, however it is possible to specify the starting value by adding a parameter: `range(2, 6)`, which means values from 2 to 6 (but not including 6).

```
for x in range(2, 6):
  print(x)
```

- The `range()` function defaults to increment the sequence by 1, however it is possible to specify the increment value by adding a third parameter:
  `range(2, 30, 3)`

```
for x in range(2, 30, 3):
  print(x)
```

Python
Functions

- A function is a block of code which only runs when it is called.
- You can pass data, known as parameters, into a function.
- A function can return data as a result.

- In Python a function is defined using the def keyword.

```
def my_function():
  print("Hello from a function")
```

- To call a function, use the function name followed by parenthesis.

```
def my_function():
  print("Hello from a function")

my_function()
```

Python

Functions - Arguments

- Information can be passed into functions as arguments.
- Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma.
- The terms parameter and argument can be used for the same thing: information that is passed into a function.
- The following example has a function with one argument (fname). When the function is called, we pass along a first name, which is used inside the function to print the full name.

```
def my_function(fname):
  print(fname + " Olsson")

my_function("Mikael")
my_function("Rasmus")
```

Python

Functions - Arguments

- By default, a function must be called with the correct number of arguments. Meaning that if your function expects 2 arguments, you have to call the function with 2 arguments, not more, and not less.

```
def my_function(fname, lname):
  print(fname + " " + lname)

my_function("Emil", ”Refsnes")
```

- If you do not know how many arguments that will be passed into your function, add a * before the parameter name in the function definition.

- This way the function will receive a tuple of arguments, and can access the items accordingly.

```
def my_function(*kids):
  print("The youngest child is " +
kids[-1])

my_function("Emil", "Tobias", "Linus")
```

Python
Functions - Arguments

- You can also send arguments with the key = value syntax.
- This way the order of the arguments does not matter.

```
def my_function(child3, child2,
child1):
  print("The youngest child is " +
child3)

my_function(child1 = "Emil", child2
= "Tobias", child3 = "Linus")
```

```
# Arbitrary Keyword Arguments, **kwargs

def my_function(**kid):
  print("His last name is " +
kid["lname"])

my_function(fname = "Tobias", lname
= "Refsnes")
```

Python

Functions - Arguments

- The following example shows how to use a default parameter value.
- If we call the function without argument, it uses the default value.

```
def my_function(country = "Norway"):
  print("I am from " + country)

my_function("Sweden")
my_function()
my_function("Brazil")
```

Python

Functions - Arguments

- You can send any data types of argument to a function (string, number, list, dictionary etc.), and it will be treated as the same data type inside the function.

- E.g. if you send a List as an argument, it will still be a List when it reaches the function.

```
def my_function(food):
  for x in food:
    print(x)

fruits = ["apple", "banana", "cherry"]
my_function(fruits)
```

Python

Functions - Return values

- To let a function return a value, use the return statement.

```
def my_function(x):
  return 5 * x

print(my_function(3))
print(my_function(5))
```

Python

Functions - Return values

- Function definitions cannot be empty, but if you for some reason have a function definition with no content, put in the pass statement to avoid getting an error.

```
def myfunction():
  pass
```

Python
Recursion

- Python also accepts function recursion, which means a defined function can call itself.
- Recursion is a common mathematical and programming concept. It means that a function calls itself. This has the benefit of meaning that you can loop through data to reach a result.
- The developer should be very careful with recursion as it can be quite easy to slip into writing a function which never terminates, or one that uses excess amounts of memory or processor power. However, when written correctly recursion can be a very efficient and mathematically-elegant approach to programming.
- In this example, tri_recursion() is a function that we have defined to call itself ("recurse"). We use the k variable as the data, which decrements (-1) every time we recurse. The recursion ends when the condition is not greater than 0 (i.e. when it is 0).
- To a new developer it can take some time to work out how exactly this works, best way to find out is by testing and modifying it.

```
def tri_recursion(k):
  if(k > 0):
    result = k + tri_recursion(k - 1)
    print(result)
  else:
    result = 0
  return result

print("\n\nRecursion Example Results")
tri_recursion(6)
```

Exercise

Guess number

- Let the computer pick a random number and let the user guess what number it is. Give the user feedback if the number is too low or too high. The game ends when the user guesses the right word. Print out how many guesses it took.

- Areas: input, casting, conditions, loops

Exercise

Hangman

- Hangman is a paper and pencil guessing game for two or more players. One player thinks of a word, phrase or sentence and the other(s) tries to guess it by suggesting letters within a certain number of guesses.
- https://en.wikipedia.org/wiki/Hangman_(game)
- Areas: lists, conditions, loops

Exercise

FizzBuzz

- Write a program that prints the numbers from 1 to 100. But for multiples of three print “Fizz” instead of the number and for the multiples of five print “Buzz”. For numbers which are multiples of both three and five print “FizzBuzz”.

- Areas: Loops, conditions

Exercise

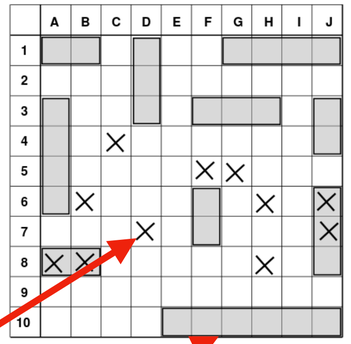
Interest

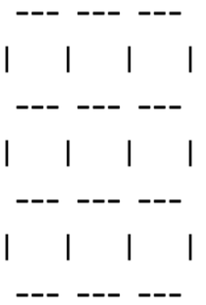
- Write a program that calculates the interest for a saving over a few years.
- Areas: Recursion

Saving	Year	Interest 10%	Sum
10 000,00 kr	1	1 000,00 kr	11 000,00 kr
11 000,00 kr	2	1 100,00 kr	12 100,00 kr
12 100,00 kr	3	1 210,00 kr	13 310,00 kr
13 310,00 kr	4	1 331,00 kr	14 641,00 kr

Afternoon exercise

Battleship





- Create some kind of collection to keep track of a game board for battleship (10 x 10 squares).
- Write a function that draws your game board. You can see a tic tac toe version here, but your map could look anyway you want.
- Add means of keeping tracks of a few boats of different size (grey areas in pictures). This could (but doesn't have to) mean a key for each square.
- Add functions for adding boats to your game board.
- Change your draw function to indicate if a boat is placed in the square, perhaps by writing a ”B”?
- Create a shoot function that takes two parameters and sets a value in the corresponding square in your game board collection. For instance shoot(”D”, 7) to indicate a shot here.
- Change the draw function to print out shots, for instance with X.
- Keep track of misses on the board.
- Draw misses by printing out, for instance, ”M”.

Python
Övningar

- https://www.w3schools.com/python/exercise.asp
- https://www.w3resource.com/python-exercises/