3. Python types
List operations - sorting items in a list

You can sort a list very simply in python. Later in the course we will show you how to use custom functions to perform sorts, but for know...

- \( l = [1, 1, 23, 87, 4, 93, -1, 1, 3, 23] \)
- \( l = l.sort() \)
- \( l = l.reverse() \)
- \( \text{print} \ l \)
You can sort a list very simply in python. Later in the course we will show you how to use custom functions to perform sorts, but for know...

```python
l = [1, 1, 23, 87, 4, 93, -1, 1, 3, 23]
l = l.sort()
print(l)
```

The `sort()` method by default sorts in ascending order. To obtain a descending order you can...

```python
l = [1, 1, 23, 87, 4, 93, -1, 1, 3, 23]
l = l.sort()
l = l.reverse()
print(l)
```
Occasionally you may want to split a string into a list of words.

- \( s = \text{`The quick brown fox jumps over the lazy dog.'} \)
- \( l = s\text{.split()} \)
- print \( l \)
3. Python types
List operations - converting a list into a string and vice versa

Occasionally you may want to split a string into a list of words.

► s = ‘The quick brown fox jumps over the lazy dog.’
► l = s.split()
► print l

Then you may want to convert the contents of a list back into a string.

► newStr = ‘ ’.join(l)
► print newStr
Occasionally you may want to split a string into a list of words.

```python
s = 'The quick brown fox jumps over the lazy dog.'
l = s.split()
print l
```

Then you may want to convert the contents of a list back into a string.

```python
newStr = ' '.join(l)
print newStr
```

Notes:

- The `join()` method will not cast values into strings. Therefore, the contents of the list must all be strings.
- The contents between the two quotes in front of `join()` can be any string. This value will be used to join all of the elements of the list.
3. Practice with lists

Write some code that performs the following task of merging two lists of numbers, creating a new list of numbers, in ascending order and each non-unique number is only represented once.

Two lists:

- [43, 900, 34.3, 2, -1]
- [5, 98, -23, 0, 2, 43]

How many different ways can you imagine using what you currently know to accomplish this?
A comprehension is a fast method for creating a list with certain contents.

They are very flexible! As you will see on the next slide.

- `list2 = [x for x in range(0,33,1)]`

Let’s break this into its components...

*result* = [*transform* *iteration*]
3. Python types

List comprehensions

But what is you only want every 3rd number between 0 and 33? There is an additional component of a list comprehension. It is called a filter.

\[ \text{result} = \text{[transform iteration]} \]

It works like this:

\[ \text{list2} = [x \text{ for } x \text{ in range}(0,33,1) \text{ if } x \% 3 == 0] \]

Let's work through this syntax.
How to we quickly obtain a 3x3 matrix in which the first row is 1,2,3, the second is 3,4,5, and the third is 6,7,8. Use list comprehensions to accomplish this.
How to quickly obtain a 3x3 matrix in which the first row is 1,2,3, the second is 3,4,5, and the third is 6,7,8. Use list comprehensions to accomplish this.

Answer...

```python
l = [[i for i in range(1,4,1)], [j for j in range(4,7,1)], [k for k in range(7,10,1)]]
```

Now use the list comprehension to create a new list (matrix) that is transposed.

Answer...

```python
l = [[r[i] for r in l] for i in range(4)]
```
3. Practice with list comprehensions

Nested list comprehensions

How to quickly obtain a 3x3 matrix in which the first row is 1,2,3, the second is 3,4,5, and the third is 6,7,8. Use list comprehensions to accomplish this.

Answer...

- \( l = [[i \text{ for } i \text{ in } \text{range}(1,4,1)], [j \text{ for } j \text{ in } \text{range}(4,7,1)], [k \text{ for } k \text{ in } \text{range}(7,10,1)]] \)

Now using the list \( l \) use a list comprehension to create a new list (matrix) that is transposed.

Answer...

- \( l = [[r[i] \text{ for } r \text{ in } l] \text{ for } i \text{ in } \text{range}(4)] \)
Tuples are another sequence class similar to a list in containing any other type and in how they are addressed. However, they are not mutable like lists. Values can be assigned to a tuple in the following ways:

- \( x = 1, 2, 'green', [7,8,9] \)
- \( \text{print } x \)

Alternatively,

- \( x = (1, 2, 'green', [7,8,9]) \)
- \( \text{print } x \)
To retrieve something you simply do the following...

- \(x = (1, 2, \text{'green'}, [7,8,9])\)
- \(w = x[0]\)
- \(z = x[2]\)
- \(y = x[3]\)
- `print w, z, y`

However, be careful of the following behavior!

- \(x = (1, 2, \text{'green'}, [7,8,9])\)
- \(y = x[3]\)
- `print y`
- \(y[0]=10\)
- `print y`
- `print x`
If you want to retrieve a copy of a list or dictionary you must do the following...

- \( x = (1, 2, \text{'green'}, [7,8,9]) \)
- \( y = \text{list}(x[3]) \)
- \( \text{print } y \)

Now when you change \( y \) only the elements of \( y \) change, not \( x \).
If you want to retrieve a copy of a list or dictionary you must do the following...

- \( x = (1, 2, 'green', [7,8,9]) \)
- \( y = \text{list}(x[3]) \)
- \( \text{print } y \)

Now when you change \( y \) only the elements of \( y \) change, not \( x \).

While they are not mutable themselves any mutable type contained within them can be changed in place. E.g.,

- \( x = (1, 2, 'green', [7,8,9]) \)
- \( y = x[3][0] \)
- \( \text{print } y \)
You can also retrieve multiple elements simultaneously...

- `x = (1, 2, 'green', [7,8,9])`
- `w,y,z = x[0:3]`
- `print w, y, z`

This is called *unpacking* a tuple.

We will see later on that tuples are very useful for returning information from functions.
Dictionaries are mutable, unordered, containers in which a value is associated with a key.

These are often referred to as key-valued containers. There is always 1 value for every key. Keys must be unique!
Dictionaries are mutable, unordered, containers in which a \textit{value} is associated with a \textit{key}.

These are often referred to as \textit{key-valued} containers. There is always 1 \textit{value} for every \textit{key}. Keys must be unique!

For example, you may want to store the name (the \textit{key}) of a person and her or his age (the \textit{value}). How do we do this?

```python
x = {'Jon':42}
print x
```

Alternatively you can create an empty dictionary and then add the key-value to it

```python
x = {}
x['Jon'] = 42
x['Fabienne'] = '?'
print x
```
Dictionaries are mutable, unordered, containers in which a value is associated with a key.

These are often referred to as key-valued containers. There is always 1 value for every key. Keys must be unique!

For example, you may want to store the name (the key) of a person and her or his age (the value). How do we do this?

```python
x = {'Jon':42}
print x
```
Dictionaries are mutable, unordered, containers in which a value is associated with a key.

These are often referred to as key-valued containers. There is always 1 value for every key. Keys must be unique!

For example, you may want to store the name (the key) of a person and her or his age (the value). How do we do this?

```
▶ x = {‘Jon’:42}
▶ print x
```

Alternatively you can create an empty dictionary and then add the key-value to it

```
▶ x = {}
▶ x[‘Jon’] = 42
▶ x[‘Fabienne’] = ’?’
▶ print x
```
3. Python types
Dictionaries - facts about keys and values

Facts about *keys*:

1. *Keys* can only be strings or a number.
2. *Keys* must be unique. I.e., you can not have two *keys* with the same name but different values.
3. *Keys* themselves are not mutable, i.e., they can not be changed.
4. *Keys* and the associated *value* can be removed.

Facts about *values*:

1. *Values* can be of any type, even another dictionary.
2. *Values* can be removed (using the `None` type), modified, or replaced with another type.

Now let’s get familiar with dictionaries.
Five different ways of making five identical dictionaries
Five different ways of making five identical dictionaries

- a = dict(one=1, two=2, three=3)
3. Python types

Dictionaries - more ways to add items

Five different ways of making five identical dictionaries

- $a = \text{dict(one=1, two=2, three=3)}$
- $b = \{'\text{one}': 1, '\text{two}': 2, '\text{three}': 3\}$
Five different ways of making five identical dictionaries

- $a = \text{dict}(\text{one}=1, \text{two}=2, \text{three}=3)$
- $b = \{\text{`one': 1, `two': 2, `three': 3}\}$
- $c = \text{dict}(\text{zip([`one', `two', `three'], [1, 2, 3])})$
Five different ways of making five identical dictionaries

- \( a = \text{dict}(\text{one}=1, \text{two}=2, \text{three}=3) \)
- \( b = \{\text{‘one’: 1, ‘two’: 2, ‘three’: 3}\} \)
- \( c = \text{dict}(\text{zip([‘one’, ‘two’, ‘three’], [1, 2, 3])}) \)
- \( d = \text{dict}([\text{‘two’, 2}), (\text{‘one’, 1}), (\text{‘three’, 3})]) \)
Five different ways of making five identical dictionaries

- \[a = \text{dict}(\text{one}=1, \text{two}=2, \text{three}=3)\]
- \[b = \{\text{‘one’}: 1, \text{‘two’}: 2, \text{‘three’}: 3\}\]
- \[c = \text{dict}(\text{zip([‘one’, ‘two’, ‘three’], [1, 2, 3])})\]
- \[d = \text{dict}([\text{‘two’, 2}, \text{‘one’, 1}, \text{‘three’, 3}])\]
- \[e = \text{dict}(\text{‘three’: 3, ‘one’: 1, ‘two’: 2})\]

Now let’s check that they are all the same...

```python
print a == b == c == d == e
```
Five different ways of making five identical dictionaries

- `a = dict(one=1, two=2, three=3)`
- `b = {'one': 1, 'two': 2, 'three': 3}`
- `c = dict(zip(['one', 'two', 'three'], [1, 2, 3]))`
- `d = dict([('two', 2), ('one', 1), ('three', 3)])`
- `e = dict('three': 3, 'one': 1, 'two': 2)`

Now let's check that they are all the same...

- `print a == b == c == d == e`
Start by creating the following dictionary

```
>>> a = dict(one=1, two=('red', 'green'), three=[1, 2, 3])
```
Start by creating the following dictionary

```python
a = dict(one=1, two=('red', 'green'), three=[1, 2, 3])
```

Now how do we retrieve the value for the key called ‘one’...

```python
x = a['one']
print(x)
```
Start by creating the following dictionary

```python
a = dict(one=1, two=('red', 'green'), three=[1,2,3])
```

Now how do we retrieve the value for the key called ‘one’...

```python
x = a['one']
print x
```

How do you retrieve the value called ‘green’ for the key called ‘two’...
Start by creating the following dictionary

```
    a = dict(one=1, two=('red','green'), three=[1,2,3])
```

Now how do we retrieve the *value* for the *key* called ‘one’...

```
    x = a['one']
    print x
```

How do you retrieve the *value* called ‘green’ for the *key* called ‘two’...

```
    x = a['two'][1]
    print x
```
3. Python types
Dictionaries - retrieving values

Start by creating the following dictionary

```
   a = dict(one=1, two=('red','green'), three=[1,2,3])
```

Now how do we retrieve the *value* for the *key* called ‘one’...

```
   x = a['one']
   print x
```

How do you retrieve the *value* called ‘green’ for the *key* called ‘two’...

```
   x = a['two'][1]
   print x
```

Take a few minutes and practice adding and retrieving values in the dictionary.
Another way of retrieving a value when you don’t know whether the key exists in the dictionary.

```
▶ a = dict(one=1, two=('red', 'green'), three=[1,2,'happy'])
▶ print a.get('two')
```
Another way of retrieving a value when you don’t know whether the key exists in the dictionary.

- `a = dict(one=1, two=('red','green'), three=[1,2,'happy'])`
- `print a.get('two')`

Why is this useful? Well first let’s look at what happens when you ask for a key that does not exist.

- `a['four']`
Another way of retrieving a value when you don’t know whether the key exists in the dictionary.

- `a = dict(one=1, two=('red','green'), three=[1,2,'happy'])`
- `print a.get('two')`

Why is this useful? Well first let’s look at what happens when you ask for a key that does not exist.

- `a['four']`

Let’s see what happens when you use the `get()` method.

- `a.get('four')`
Another way of retrieving a value when you don’t know whether the key exists in the dictionary.

- a = dict(one=1, two=('red','green'), three=[1,2,'happy'])
- print a.get('two')

Why is this useful? Well first let’s look at what happens when you ask for a key that does not exist.

- a['four']

Let’s see what happens when you use the get() method.

- a.get('four')

This allows you to check whether a None value is returned rather than getting a key-value error.
Changing a *value* for a *key* in a dictionary is simple.

```
▶ a = dict(one=1, two=('red','green'), three=[1,2,'happy'])
▶ print a['one']
▶ a['one'] = 100
▶ print a['one']
```

How would you change the value ‘red’ to ‘purple’?

Tuples are not mutable so you have to,

```
▶ a['two'] = ('purple','green')
▶ print a['two']
```

How would you change ‘happy’ to ‘sad’?

```
▶ a['three'][2] = 'sad'
▶ print a['three']
```
Changing a *value* for a *key* in a dictionary is simple.

```
▶ a = dict(one=1, two=('red','green'), three=[1,2,'happy'])
▶ print a['one']
▶ a['one'] = 100
▶ print a['one']
```

How would you change the value ‘red’ to ‘purple’? Tuples are not mutable so you have to,

```
▶ a['two'] = ('purple','green')
▶ print a['two']
```
3. Python types
Dictionaries - modifying values

Changing a value for a key in a dictionary is simple.

```
▶ a = dict(one=1, two=('red','green'), three=[1,2,'happy'])
▶ print a['one']
▶ a['one'] = 100
▶ print a['one']
```

How would you change the value ‘red’ to ‘purple’? Tuples are not mutable so you have to,

```
▶ a['two'] = ('purple','green')
▶ print a['two']
```

How would you change ‘happy’ to ‘sad’?
3. Python types
Dictionaries - modifying values

Changing a *value* for a *key* in a dictionary is simple.

```python
a = dict(one=1, two=('red','green'), three=[1,2,'happy'])
print a['one']
a['one'] = 100
print a['one']
```

How would you change the value ‘red’ to ‘purple’? Tuples are not mutable so you have to,

```python
a['two'] = ('purple','green')
print a['two']
```

How would you change ‘happy’ to ‘sad’?

```python
a['three'][2] = 'sad'
print a['three']
```
3. Python types

Dictionaries - removing entries (key) and values

Removing a *key-value* is also easy.

> del a[‘three’]
Removing a *key-value* is also easy.

- `del a['three']`

If you want to retain the value but remove it from the dictionary you can use the `pop()` method.

- `z = a.pop('three')`
- `print z`
- `print a`
Removing a *key-value* is also easy.

- `del a[‘three’]`

If you want to retain the value but remove it from the dictionary you can use the `pop()` method.

- `z = a.pop(‘three’)`
- `print z`
- `print a`

If you want to remove all items you,  
- `a.clear()`
- `print a`
There are three primary comparison operators for dictionaries. Two of which you have already been introduced to.

- `in`
- `not in`
- `has_key()`

Let's look at the last one. The `has_key()` checks whether a key exists returning `True` or `False`

```python
a = dict(one=1, two=('red','green'), three=[1,2,'happy'])
print a.has_key('one')
```
3. Python types

Dictionaries - iteration

There are a variety of ways of iterating over a dictionary. Most of them involve iterating over the keys.

First, using `in`

```python
a = dict(one=1, two=('red','green'), three=[1,2,'happy'])
for x in a:
    print a[x]
```

Second, by getting the keys first with sorting (in this example)

```python
a = dict(one=1, two=('red','green'), three=[1,2,'happy'])
f = a.keys()
f.sort()
for x in f:
    print a[x]
```

Third, by values

```python
for x in s.values():
    print 'Value: ', x
```
3. Python types

Dictionaries - iteration

There are a variety of ways of iterating over a dictionary. Most of them involve iterating over the keys.

First, using `in`

```python
a = dict(one=1, two=('red','green'), three=[1,2,'happy'])
for x in a:
    print a[x]
```

Second, by getting the keys first with sorting (in this example)

```python
a = dict(one=1, two=('red','green'), three=[1,2,'happy'])
f = a.keys()
f.sort()
for x in f:
    print a[x]
```
There are a variety of ways of iterating over a dictionary. Most of them involve iterating over the keys.

First, using `in`

```python
a = dict(one=1, two=('red', 'green'), three=[1, 2, 'happy'])
for x in a:
    print a[x]
```

Second, by getting the keys first with sorting (in this example)

```python
a = dict(one=1, two=('red', 'green'), three=[1, 2, 'happy'])
f = a.keys()
f.sort()
for x in f:
    print a[x]
```

Third, by values

```python
for x in s.values():
    print 'Value: ', x
```
There are two other ways of iterating over a dictionary. They use `iterkeys()` and `itervalues()`.

Here is how you use them.

```python
a = dict(one=1, two=('red', 'green'), three=[1, 2, 'happy'])
c = a.iterkeys()
for x in c:
    print a[x]
```

Now over the values.

```python
d = a.itervalues()
for x in c:
    print x
```
Pair up with somebody else and create a script file with a dictionary that contains:

1. Two *keys* – your first names.
2. And the following *values* associated with each of the *keys*:
   2.1 last name
   2.2 age
   2.3 height
   2.4 country of origin
   2.5 area of research, e.g., biology, CS, etc.

Have the script print out the information for each entry in the dictionary.

Create two separate implementations using first a list and then a dictionary to hold the associated *values*. 
1. Making a copy of a dictionary. This is important because remember that simply saying \( a = b \), where \( b \) is a dictionary does not make a copy but rather creates a new reference from \( a \) to \( b \).

\[
a = \text{dict(one=1, two=('red','green'), three=[1,2,'happy'])}
\]

\[
\text{acopy = dict(a)}
\]

\[
\text{Method 1}
\]

\[
\text{print acopy}
\]

\[
\text{bcopy = a.copy()}
\]

\[
\text{Method 2}
\]

\[
\text{print bcopy}
\]
3. Python types

Dictionaries - other tasks

1. Making a copy of a dictionary. This is important because remember that simply saying \texttt{a = b}, where \texttt{b} is a dictionary does not make a copy but rather creates a new reference from \texttt{a} to \texttt{b}.

   \begin{itemize}
   \item \texttt{a = dict(one=1, two=(‘red’, ‘green’), three=[1,2,’happy’])}
   \item \texttt{acopy = dict(a)} \# Method 1
   \item \texttt{print acopy}
   \end{itemize}
1. Making a copy of a dictionary. This is important because remember that simply saying \( a = b \), where \( b \) is a dictionary does not make a copy but rather creates a new reference from \( a \) to \( b \).

\[
\begin{align*}
& a = \text{dict(one=1, two=('red','green'), three=[1,2,'happy'])} \\
& \text{acopy = dict(a) \# Method 1} \\
& \text{print acopy} \\
& \text{bcopy = a.copy() \# Method 2} \\
& \text{print bcopy}
\end{align*}
\]
3. Python types
Dictionaries - other tasks

1. Making a copy of a dictionary. This is important because remember that simply saying `a = b`, where `b` is a dictionary does not make a copy but rather creates a new reference from `a` to `b`.

   ▶ `a = dict(one=1, two=('red','green'), three=[1,2,'happy'])`
   ▶ `acopy = dict(a) # Method 1`
   ▶ `print acopy`
   ▶ `bcopy = a.copy() # Method 2`
   ▶ `print bcopy`

2. Replacing the contents of one dictionary with another dictionary.

   ▶ `print a`
   ▶ `b = dict(green=2013, purple=2012)]`
   ▶ `print b`
   ▶ `b.update(a)`
   ▶ `print b`
3. Python types
Dictionaries - other tasks continued. . .

3. Determining the number of keys in a dictionary.
   
   - a = dict(one=1, two=('red’, ‘green’), three=[1,2,‘happy’])
   - print len(a)
3. Python types
Dictionaries - other tasks continued. . .

3. Determining the number of keys in a dictionary.
   ▶  
a = dict(one=1, two=('red','green'), three=[1,2,'happy'])
▶  
print len(a)

4. Getting a copy a dictionaries key-value pairs as a list of tuples.
   ▶  
z = a.items()
▶  
print z

Note that the returned list contains tuples with the first element (0) being the key and the second element (1) being the value.
EXERCISE: Merging dictionaries. Implement an algorithm to merge dictionaries.
A set is a collection of unordered unique elements. As you will see these are value useful for testing membership and performing mathematical set operations.

Set elements, unlike lists, are immutable and can not be changed once created. You can add and remove items from sets.
3. Python types

Sets

A set is a collection of unordered unique elements. As you will see these are value useful for testing membership and performing mathematical set operations.

Set elements, unlike lists, are immutable and can not be changed once created. You can add and remove items from sets.

To create a set you simply do the following:

```python
▶ s = set([1,2,3,4,5])
▶ print s
```
3. Python types

Sets

A set is a collection of unordered unique elements. As you will see these are very useful for testing membership and performing mathematical set operations.

Set elements, unlike lists, are immutable and cannot be changed once created. You can add and remove items from sets.

To create a set you simply do the following:

- \( s = \text{set}([1,2,3,4,5]) \)
- \( \text{print} \ s \)

The `set()` constructor takes one argument that is iterable, e.g., a list, a string, another set, etc. See what happens when you pass a string
A set is a collection of unordered unique elements. As you will see these are useful for testing membership and performing mathematical set operations.

Set elements, unlike lists, are immutable and can not be changed once created. You can add and remove items from sets.

To create a set you simply do the following:

1. `s = set([1,2,3,4,5])`
2. `print s`

The `set()` constructor takes one argument that is iterable, e.g., a list, a string, another set, etc. See what happens when you pass a string:

1. `s = set('green')`
2. `print s`
Create the following two sets:

- \( s = \text{set}([1,2,3,4,5,\text{\textquoteleft}green\text{\textquoteleft}]) \)
- \( t = \text{set}([3,2,10,-1,\text{\textquoteleft}green\text{\textquoteleft},\text{\textquoteleft}red\text{\textquoteleft}]) \)
Create the following two sets:

- \( s = \text{set}([1, 2, 3, 4, 5, \text{\textquoteleft\textquoteleft}green\textquoteright\textquoteright]) \)
- \( t = \text{set}([3, 2, 10, -1, \text{\textquoteleft\textquoteleft}green\textquoteright\textquoteright, \text{\textquoteleft\textquoteleft}red\textquoteright\textquoteright]) \)

You can perform the following operations:

- \( \text{len}(s) \) cardinality of set \( s \)
Create the following two sets:

- \( s = \text{set}([1,2,3,4,5,\text{‘green’}]) \)
- \( t = \text{set}([3,2,10,-1,\text{‘green’},\text{‘red’}]) \)

You can perform the following operations:

- \( \text{len}(s) \)  cardinality of set \( s \)
- \( x \text{ in } s \) test \( x \) for membership in \( s \)
3. Python types
Sets - set operations

Create the following two sets:

- \( s = \text{set}([1,2,3,4,5,('green')]) \)
- \( t = \text{set}([3,2,10,-1,'green','red']) \)

You can perform the following operations:

- \( \text{len}(s) \) cardinality of set \( s \)
- \( x \text{ in } s \) test \( x \) for membership in \( s \)
- \( x \text{ not in } s \) test \( x \) for non-membership in \( s \)
3. Python types
Sets - set operations

Create the following two sets:

- \( s = \text{set}([1,2,3,4,5,'green']) \)
- \( t = \text{set}([3,2,10,-1,'green','red']) \)

You can perform the following operations:

- \( \text{len}(s) \): cardinality of set \( s \)
- \( x \text{ in } s \): test \( x \) for membership in \( s \)
- \( x \text{ not in } s \): test \( x \) for non-membership in \( s \)
- \( s.\text{issubset}(t) \): test whether every element in \( s \) is in \( t \)
Create the following two sets:

- \( s = \text{set}([1,2,3,4,5,\text{('green')})] \)
- \( t = \text{set}([3,2,10,-1,\text{('green','red')})] \)

You can perform the following operations:

- \( \text{len}(s) \): cardinality of set \( s \)
- \( x \text{ in } s \): test \( x \) for membership in \( s \)
- \( x \text{ not in } s \): test \( x \) for non-membership in \( s \)
- \( s.\text{issubset}(t) \): \( s \subseteq t \) test whether every element in \( s \) is in \( t \)
- \( s.\text{issuperset}(t) \): \( s \supseteq t \) test whether every element in \( t \) is in \( s \)
Create the following two sets:

- $s = \text{set}([1,2,3,4,5,\text{\textquotesingle}green\text{\textquotesingle}])$
- $t = \text{set}([3,2,10,-1,\text{\textquotesingle}green\text{\textquotesingle},\text{\textquotesingle}red\text{\textquotesingle}])$

You can perform the following operations:

- `len(s)` cardinality of set $s$
- `x in s` test $x$ for membership in $s$
- `x not in s` test $x$ for non-membership in $s$
- `s.issubset(t)` $s \subseteq t$ test whether every element in $s$ is in $t$
- `s.issuperset(t)` $s \supseteq t$ test whether every element in $t$ is in $s$
- `s.union(t)` $s \mid t$ new set with elements from both $s$ and $t$
3. Python types
Sets - set operations

Create the following two sets:

- \( s = \text{set}([1,2,3,4,5,('green')]) \)
- \( t = \text{set}([3,2,10,-1,('green','red')]) \)

You can perform the following operations:

- `len(s)` cardinality of set \( s \)
- `x in s` test \( x \) for membership in \( s \)
- `x not in s` test \( x \) for non-membership in \( s \)
- `s.issubset(t)` \( s \subseteq t \) test whether every element in \( s \) is in \( t \)
- `s.issuperset(t)` \( s \supseteq t \) test whether every element in \( t \) is in \( s \)
- `s.union(t)` \( s \cup t \) new set with elements from both \( s \) and \( t \)
- `s.intersection(t)` \( s \cap t \) new set with elements common to \( s \) and \( t \)
Create the following two sets:

- \( s = \text{set}([1,2,3,4,5,\text{\textquoteleft}green\textquoteright]) \)
- \( t = \text{set}([3,2,10,-1,\text{\textquoteleft}green\textquoteright,\text{\textquoteleft}red\textquoteright]) \)

You can perform the following operations:

- \( \text{len}(s) \): cardinality of set \( s \)
- \( x \text{ in } s \): test \( x \) for membership in \( s \)
- \( x \text{ not in } s \): test \( x \) for non-membership in \( s \)
- \( s.\text{issubset}(t) \): \( s \subseteq t \) test whether every element in \( s \) is in \( t \)
- \( s.\text{issuperset}(t) \): \( s \supseteq t \) test whether every element in \( t \) is in \( s \)
- \( s.\text{union}(t) \): \( s | t \) new set with elements from both \( s \) and \( t \)
- \( s.\text{intersection}(t) \): \( s \& t \) new set with elements common to \( s \) and \( t \)
- \( s.\text{difference}(t) \): \( s - t \) new set with elements in \( s \) but not in \( t \)
You may in your program want to check whether a certain variable is of a certain type.

- `s = 'green'`
- `isinstance(s, str)`
- `isinstance(s, dict)`