Dictionaries and Sets

Dictionaries and Sets

Hash maps and dynamic binding

Indexing by words

Indexing by words

So far, we've used integer indexes to access individual entries in lists, tuples, and strings.

Indexing by words

So far, we've used integer indexes to access individual entries in lists, tuples, and strings.

```
>>> animals = ["cow", "pig", "horse", "dog"]
```

Indexing by words

So far, we've used integer indexes to access individual entries in lists, tuples, and strings.

```
>>> animals = ["cow", "pig", "horse", "dog"]
>>> print(animals[0])
```

Indexing by words

So far, we've used integer indexes to access individual entries in lists, tuples, and strings.

```
>>> animals = ["cow", "pig", "horse", "dog"]
>>> print(animals[0])
cow
```

Indexing by words

So far, we've used integer indexes to access individual entries in lists, tuples, and strings.

```
>>> animals = ["cow", "pig", "horse", "dog"]
>>> print(animals[0])
cow
```

But what if we wanted to index by string instead?

Hash Maps

Hash Maps

To index by string, we use a function called a *hash map*

Hash Maps

To index by string, we use a function called a *hash map*

$$H(s) = x$$

Hash Maps

To index by string, we use a function called a *hash map*

$$H(s) = x$$

Takes *strings* as input,

Hash Maps

To index by string, we use a function called a *hash map*

$$H(s) = x$$

Takes *strings* as input, and outputs *integers*

Hash Maps

Hash Maps

You can then use those integer outputs to index a list

Hash Maps

You can then use those integer outputs to index a list

`"Tyler"`

Hash Maps

You can then use those integer outputs to index a list

`"Tyler"` $\rightarrow H$

Hash Maps

You can then use those integer outputs to index a list

$\texttt{"Tyler"} \rightarrow H \rightarrow 234$

Hash Maps

You can then use those integer outputs to index a list

"Tyler" $\rightarrow H \rightarrow$ 234 $\rightarrow$ `myList[234]`

Hash Maps

You can then use those integer outputs to index a list

`"Tyler"` $\rightarrow H \rightarrow$ 234 $\rightarrow$ `myList[234]`

How to create a good hash map is complicated

Hash Maps

You can then use those integer outputs to index a list

`"Tyler"` $\rightarrow H \rightarrow$ 234 $\rightarrow$ `myList[234]`

How to create a good hash map is complicated

It involves using the string's ASCII codes and clever mathematics to avoid *collision*

Hash Maps

You can then use those integer outputs to index a list

`"Tyler"` $\rightarrow H \rightarrow 234 \rightarrow$ `myList[234]`

How to create a good hash map is complicated

It involves using the string's ASCII codes and clever mathematics to avoid *collision*

Python gives you access to hash maps using the *dictionary* type

Dicts

Dicts

Create an empty dictionary using the following syntax:

Dicts

Create an empty dictionary using the following syntax:

```
>>> myDict = {}
```

Dicts

Create an empty dictionary using the following syntax:

```
>>> myDict = {}
```

Create a dictionary of values by using the following syntax:

Dicts

Create an empty dictionary using the following syntax:

```
>>> myDict = {}
```

Create a dictionary of values by using the following syntax:

```
>>> myDict = {"Entry1":  1.0,
              "Entry2":  2.0,
              "Entry3":  3.0}
```

Dicts

Create an empty dictionary using the following syntax:

```
>>> myDict = {}
```

Create a dictionary of values by using the following syntax:

```
>>> myDict = {"Entry1":  1.0,
              "Entry2":  2.0,
              "Entry3":  3.0}
```

The labels ("Entry1", "Entry2", and "Entry3") are called *keys*, and the corresponding items (1.0, 2.0, 3.0) are called *values*

Dicts continued

Dicts continued

Index a dict using the keys:

Dicts continued

Index a dict using the keys:

```
>>> print(myDict["Entry1"])
```

Dicts continued

Index a dict using the keys:

```
>>> print(myDict["Entry1"])
1.0
```

Dicts continued

Index a dict using the keys:

```
>>> print(myDict["Entry1"])
1.0
```

Dicts are mutable. Update values by assignment:

Dicts continued

Index a dict using the keys:

```
>>> print(myDict["Entry1"])
1.0
```

Dicts are mutable. Update values by assignment:

```
>>> myDict["Entry2"] = 2.5
```

Dicts continued

Index a dict using the keys:

```
>>> print(myDict["Entry1"])
1.0
```

Dicts are mutable. Update values by assignment:

```
>>> myDict["Entry2"] = 2.5
```

Add new entries to a dict by assigning to a nonexisted key:

Dicts continued

Index a dict using the keys:

```
>>> print(myDict["Entry1"])
1.0
```

Dicts are mutable. Update values by assignment:

```
>>> myDict["Entry2"] = 2.5
```

Add new entries to a dict by assigning to a nonexisted key:

```
>>> myDict["Tyler"] = "Chang"
```

Dicts methods

Dicts methods

Just like lists and strings, dicts have several built-in methods.

Dicts methods

Just like lists and strings, dicts have several built-in methods.

The most useful is the `dict.keys()` method:

Dicts methods

Just like lists and strings, dicts have several built-in methods.

The most useful is the `dict.keys()` method:

```
>>> print(myDict.keys())
```

Dicts methods

Just like lists and strings, dicts have several built-in methods.

The most useful is the `dict.keys()` method:

```
>>> print(myDict.keys())
["Entry1", "Entry2", "Entry3", "Tyler"]
```

Dicts methods

Just like lists and strings, dicts have several built-in methods.

The most useful is the `dict.keys()` method:

```
>>> print(myDict.keys())
["Entry1", "Entry2", "Entry3", "Tyler"]

>>> for key in myDict.keys():
...     print(myDict[key])
```

Dicts methods

Just like lists and strings, dicts have several built-in methods.

The most useful is the `dict.keys()` method:

```
>>> print(myDict.keys())
["Entry1", "Entry2", "Entry3", "Tyler"]

>>> for key in myDict.keys():
...     print(myDict[key])
1.0
```

Dicts methods

Just like lists and strings, dicts have several built-in methods.

The most useful is the `dict.keys()` method:

```
>>> print(myDict.keys())
["Entry1", "Entry2", "Entry3", "Tyler"]

>>> for key in myDict.keys():
...     print(myDict[key])
1.0
2.5
```

Dicts methods

Just like lists and strings, dicts have several built-in methods.

The most useful is the `dict.keys()` method:

```
>>> print(myDict.keys())
["Entry1", "Entry2", "Entry3", "Tyler"]

>>> for key in myDict.keys():
...     print(myDict[key])
1.0
2.5
3.0
```

Dicts methods

Just like lists and strings, dicts have several built-in methods.

The most useful is the `dict.keys()` method:

```
>>> print(myDict.keys())
["Entry1", "Entry2", "Entry3", "Tyler"]

>>> for key in myDict.keys():
...     print(myDict[key])
1.0
2.5
3.0
Chang
```

Dicts methods

Just like lists and strings, dicts have several built-in methods.

The most useful is the `dict.keys()` method:

```
>>> print(myDict.keys())
["Entry1", "Entry2", "Entry3", "Tyler"]

>>> for key in myDict.keys():
...     print(myDict[key])
1.0
2.5
3.0
Chang
```

Read more about dict methods in the Python docs:
`https://docs.python.org/3/library/stdtypes.html#mapping-types-dict`

Concept Check!

Consider an empty dict:

```
myDict = {}
```

What is the value of `myDict` running the following commands:

1. `myDict["name"] = "Marky"`
2. `myDict["ID"] = 54321`
3. `myDict["name"] = "Zarky"`
4. `myDict["password"] = "what"`

Concept Check!

Consider an empty dict:

`myDict = {}`

What is the value of `myDict` running the following commands:

1. `myDict["name"] = "Marky"` `{"name": "Marky"}`
2. `myDict["ID"] = 54321`
3. `myDict["name"] = "Zarky"`
4. `myDict["password"] = "what"`

Concept Check!

Consider an empty dict:

```
myDict = {}
```

What is the value of `myDict` running the following commands:

1. `myDict["name"] = "Marky"`

   ```
   {"name":  "Marky"}
   ```

2. `myDict["ID"] = 54321`

   ```
   {"name":  "Marky",
    "ID":  54321}
   ```

3. `myDict["name"] = "Zarky"`

4. `myDict["password"] = "what"`

Concept Check!

Consider an empty dict:

```
myDict = {}
```

What is the value of `myDict` running the following commands:

1. `myDict["name"] = "Marky"`

```
{"name":  "Marky"}
```

2. `myDict["ID"] = 54321`

```
{"name":  "Marky",
 "ID":  54321}
```

3. `myDict["name"] = "Zarky"`

```
{"name":  "Zarky",
 "ID":  54321}
```

4. `myDict["password"] = "what"`

Concept Check!

Consider an empty dict:

```
myDict = {}
```

What is the value of `myDict` running the following commands:

1. `myDict["name"] = "Marky"`

```
{"name":  "Marky"}
```

2. `myDict["ID"] = 54321`

```
{"name":  "Marky",
 "ID":  54321}
```

3. `myDict["name"] = "Zarky"`

```
{"name":  "Zarky",
 "ID":  54321}
```

4. `myDict["password"] = "what"`

```
{"name":  "Zarky",
 "ID":  54321,
 "password":  "what"}
```

Dynamic binding

Dynamic binding

Assigning dict key names at runtime is an example of *dynamic binding*:

Dynamic binding

Assigning dict key names at runtime is an example of *dynamic binding*:

Pick variable name at runtime:

Dynamic binding

Assigning dict key names at runtime is an example of *dynamic binding*:

Pick variable name at runtime:

```
>>> database = {}
```

Dynamic binding

Assigning dict key names at runtime is an example of *dynamic binding*:

Pick variable name at runtime:

```
>>> database = {}
>>> nextKey = input("Enter a name to add:  ")
```

Dynamic binding

Assigning dict key names at runtime is an example of *dynamic binding*:

Pick variable name at runtime:

```
>>> database = {}
>>> nextKey = input("Enter a name to add:  ")
Enter a name to add:
```

Dynamic binding

Assigning dict key names at runtime is an example of *dynamic binding*:

Pick variable name at runtime:

```
>>> database = {}
>>> nextKey = input("Enter a name to add:  ")
Enter a name to add:  Tyler
```

Dynamic binding

Assigning dict key names at runtime is an example of *dynamic binding*:

Pick variable name at runtime:

```
>>> database = {}
>>> nextKey = input("Enter a name to add:  ")
Enter a name to add:  Tyler
>>> nextVal = input(f"What is {nextKey}'s position?  ")
```

Dynamic binding

Assigning dict key names at runtime is an example of *dynamic binding*:

Pick variable name at runtime:

```
>>> database = {}
>>> nextKey = input("Enter a name to add:  ")
Enter a name to add:  Tyler
>>> nextVal = input(f"What is {nextKey}'s position?  ")
What is Tyler's position?
```

Dynamic binding

Assigning dict key names at runtime is an example of *dynamic binding*:

Pick variable name at runtime:

```
>>> database = {}
>>> nextKey = input("Enter a name to add:  ")
Enter a name to add:  Tyler
>>> nextVal = input(f"What is {nextKey}'s position?  ")
What is Tyler's position?  instructor
```

Dynamic binding

Assigning dict key names at runtime is an example of *dynamic binding*:

Pick variable name at runtime:

```
>>> database = {}
>>> nextKey = input("Enter a name to add:  ")
Enter a name to add:  Tyler
>>> nextVal = input(f"What is {nextKey}'s position?  ")
What is Tyler's position?  instructor
database[nextKey] = nextVal
```

Dynamic binding

Assigning dict key names at runtime is an example of *dynamic binding*:

Pick variable name at runtime:

```
>>> database = {}
>>> nextKey = input("Enter a name to add:  ")
Enter a name to add:  Tyler
>>> nextVal = input(f"What is {nextKey}'s position?  ")
What is Tyler's position?  instructor
database[nextKey] = nextVal
>>> access = input("Enter a person to access:  ")
```

Dynamic binding

Assigning dict key names at runtime is an example of *dynamic binding*:

Pick variable name at runtime:

```
>>> database = {}
>>> nextKey = input("Enter a name to add:  ")
Enter a name to add:  Tyler
>>> nextVal = input(f"What is {nextKey}'s position?  ")
What is Tyler's position?  instructor
database[nextKey] = nextVal
>>> access = input("Enter a person to access:  ")
Enter a person to access:
```

Dynamic binding

Assigning dict key names at runtime is an example of *dynamic binding*:

Pick variable name at runtime:

```
>>> database = {}
>>> nextKey = input("Enter a name to add:  ")
Enter a name to add:  Tyler
>>> nextVal = input(f"What is {nextKey}'s position?  ")
What is Tyler's position?  instructor
database[nextKey] = nextVal
>>> access = input("Enter a person to access:  ")
Enter a person to access:  Tyler
```

Dynamic binding

Assigning dict key names at runtime is an example of *dynamic binding*:

Pick variable name at runtime:

```
>>> database = {}
>>> nextKey = input("Enter a name to add:  ")
Enter a name to add:  Tyler
>>> nextVal = input(f"What is {nextKey}'s position?  ")
What is Tyler's position?  instructor
database[nextKey] = nextVal
>>> access = input("Enter a person to access:  ")
Enter a person to access:  Tyler
>>> print(f"position of {access} is {database[access]}")
```

Dynamic binding

Assigning dict key names at runtime is an example of *dynamic binding*:

Pick variable name at runtime:

```
>>> database = {}
>>> nextKey = input("Enter a name to add:  ")
Enter a name to add:  Tyler
>>> nextVal = input(f"What is {nextKey}'s position?  ")
What is Tyler's position?  instructor
database[nextKey] = nextVal
>>> access = input("Enter a person to access:  ")
Enter a person to access:  Tyler
>>> print(f"position of {access} is {database[access]}")
position of Tyler is instructor
```

Sets

Sets

Sets allow you to create unordered lists without repetition:

Sets

Sets allow you to create unordered lists without repetition:

Create a set by using the set function

Sets

Sets allow you to create unordered lists without repetition:

Create a set by using the set function

```
>>> x = set()
```

Sets

Sets allow you to create unordered lists without repetition:

Create a set by using the set function

```
>>> x = set()
>>> x = set([1, 2, 3, 4])
```

Sets

Sets allow you to create unordered lists without repetition:

Create a set by using the set function

```
>>> x = set()
>>> x = set([1, 2, 3, 4])
```

Add items to the set using the `add()` function

Sets

Sets allow you to create unordered lists without repetition:

Create a set by using the set function

```
>>> x = set()
>>> x = set([1, 2, 3, 4])
```

Add items to the set using the `add()` function

```
>>> x.add(5)
```

Sets

Sets allow you to create unordered lists without repetition:

Create a set by using the set function

```
>>> x = set()
>>> x = set([1, 2, 3, 4])
```

Add items to the set using the `add()` function

```
>>> x.add(5)
>>> print(x)
```

Sets

Sets allow you to create unordered lists without repetition:

Create a set by using the set function

```
>>> x = set()
>>> x = set([1, 2, 3, 4])
```

Add items to the set using the `add()` function

```
>>> x.add(5)
>>> print(x)
{1, 2, 3, 4, 5}
```

Sets

Sets

Sets are unordered, so you can't index them to access individual items:

Sets

Sets are unordered, so you can't index them to access individual items:

```
>>> x = set()
```

Sets

Sets are unordered, so you can't index them to access individual items:

```
>>> x = set()
>>> x = set([1, 2, 3, 4])
```

Sets

Sets are unordered, so you can't index them to access individual items:

```
>>> x = set()
>>> x = set([1, 2, 3, 4])
>>> x[0]
```

Sets

Sets are unordered, so you can't index them to access individual items:

```
>>> x = set()
>>> x = set([1, 2, 3, 4])
>>> x[0]
TyperError:  "set" object is not subscriptable
```

Sets

Sets are unordered, so you can't index them to access individual items:

```
>>> x = set()
>>> x = set([1, 2, 3, 4])
>>> x[0]
TyperError:  "set" object is not subscriptable
```

They also don't store duplicate items:

Sets

Sets are unordered, so you can't index them to access individual items:

```
>>> x = set()
>>> x = set([1, 2, 3, 4])
>>> x[0]
TyperError:  "set" object is not subscriptable
```

They also don't store duplicate items:

```
>>> x = set([1, 2, 3, 4])
```

Sets

Sets are unordered, so you can't index them to access individual items:

```
>>> x = set()
>>> x = set([1, 2, 3, 4])
>>> x[0]
TyperError:  "set" object is not subscriptable
```

They also don't store duplicate items:

```
>>> x = set([1, 2, 3, 4])
>>> print(x)
```

Sets

Sets are unordered, so you can't index them to access individual items:

```
>>> x = set()
>>> x = set([1, 2, 3, 4])
>>> x[0]
TyperError:  "set" object is not subscriptable
```

They also don't store duplicate items:

```
>>> x = set([1, 2, 3, 4])
>>> print(x)
{1, 2, 3, 4}
```

Sets

Sets are unordered, so you can't index them to access individual items:

```
>>> x = set()
>>> x = set([1, 2, 3, 4])
>>> x[0]
TyperError:  "set" object is not subscriptable
```

They also don't store duplicate items:

```
>>> x = set([1, 2, 3, 4])
>>> print(x)
{1, 2, 3, 4}
>>> x.add(4)
```

Sets

Sets are unordered, so you can't index them to access individual items:

```
>>> x = set()
>>> x = set([1, 2, 3, 4])
>>> x[0]
TyperError:  "set" object is not subscriptable
```

They also don't store duplicate items:

```
>>> x = set([1, 2, 3, 4])
>>> print(x)
{1, 2, 3, 4}
>>> x.add(4)
>>> print(x)
```

Sets

Sets are unordered, so you can't index them to access individual items:

```
>>> x = set()
>>> x = set([1, 2, 3, 4])
>>> x[0]
TyperError:  "set" object is not subscriptable
```

They also don't store duplicate items:

```
>>> x = set([1, 2, 3, 4])
>>> print(x)
{1, 2, 3, 4}
>>> x.add(4)
>>> print(x)
{1, 2, 3, 4}
```

Sets

Sets

Use them to check unordered lists (e.g., for input validation loops):

Sets

Use them to check unordered lists (e.g., for input validation loops):

```
>>> legalValues = {"QB", "HB", "FB", "WR", "TE",
"OL"}
```

Sets

Use them to check unordered lists (e.g., for input validation loops):

```
>>> legalValues = {"QB", "HB", "FB", "WR", "TE",
"OL"}
position = ""
```

Sets

Use them to check unordered lists (e.g., for input validation loops):

```
>>> legalValues = {"QB", "HB", "FB", "WR", "TE",
"OL"}
position = ""
>>> while position not in legalValues:
```

Sets

Use them to check unordered lists (e.g., for input validation loops):

```
>>> legalValues = {"QB", "HB", "FB", "WR", "TE",
"OL"}
position = ""
>>> while position not in legalValues:
...     position = input("Enter NFL offense pos:  ")
```

Sets

Use them to check unordered lists (e.g., for input validation loops):

```
>>> legalValues = {"QB", "HB", "FB", "WR", "TE",
"OL"}
position = ""
>>> while position not in legalValues:
...     position = input("Enter NFL offense pos:  ")
```

Find more functions and methods in Python docs:
`https://docs.python.org/3/library/stdtypes.html#set`