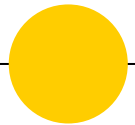
# Python object types



*“In Python, we do things with **stuff**”*

# Built-in ‘core’ data types

These objects are effectively built into the Python language.

There are many other objects defined in ‘external’ modules.

The type of an object determines what can we do with it and how it behaves:

13 + 5 = ?
“13” + “5” = ?

`type(object)`

| Object type | Example literals/creation |
|---|---|
| Numbers | `1234, 3.1415, 3+4j, 0b111, Decimal(), Fraction()` |
| Strings | `'spam', "Bob's", b'a\x01c', u'sp\xc4m'` |
| Lists | `[1, [2, 'three'], 4.5], list(range(10))` |
| Dictionaries | `{'food': 'spam', 'taste': 'yum'}, dict(hours=10)` |
| Tuples | `(1, 'spam', 4, 'U'), tuple('spam'), namedtuple` |
| Files | `open('eggs.txt'), open(r'C:\ham.bin', 'wb')` |
| Sets | `set('abc'), {'a', 'b', 'c'}` |
| Other core types | Booleans, types, `None` |
| Program unit types | Functions, modules, classes (Part IV, Part V, Part VI) |

# Methods

- Functions that are attached and act upon a specific object.
- Called with `object.method()`
- It's always useful to check which methods are available for each object:
  - https://www.w3schools.com/python/python_ref_string.asp

# Mutable vs immutable sequences

- **Immutable** sequences: strings, tuples, and bytes.
- **Mutable** sequences:lists and byte arrays. Differ from their immutable in that they can be changed after creation.

# Lists

- Ordered collections
- Mutable
- Allow duplicates
- Can contain objects of multiple types

# Dicts - Definition

- Unordered collection of data values, used to store data values like a map. It works similar to a dictionary in a real world.,
- Dictionary holds key:value pair.
    - **Key** value is provided in the dictionary to make it more optimized. Each key-value pair in a dictionary is separated by a colon :, whereas each key is separated by a ‘comma’. They must be unique and of *immutable* data type such as Strings, Integers and tuples
    - **Values** in a dictionary can be of any datatype and can be duplicated, whereas keys can’t be repeated and must be *immutable*.
- It is created by placing sequence of elements within curly {} braces, separated by ‘comma’. Dictionary can also be created by the built-in function dict(). An empty dictionary can be created by just placing to curly braces{}.

# Dicts - Creation

There are quite a few different ways to create a dictionary, so let me give you a simple example of how to create a dictionary equal to `{'A': 1, 'Z': -1}` in five different ways:

```
>>> a = dict(A=1, Z=-1)
>>> b = {'A': 1, 'Z': -1}
>>> c = dict(zip(['A', 'Z'], [1, -1]))
>>> d = dict([('A', 1), ('Z', -1)])
>>> e = dict({'Z': -1, 'A': 1})
>>> a == b == c == d == e  # are they all the same?
True  # indeed!
```

# Dicts - Example

chocolates = ['dark', 'milk', 'semi sweet'] (Which type of structure is this?)

We want to store the best brand to each type of chocolate. It is:

- 'Dark': Valor
- ‘Milk’: Nestle
- ‘Semi sweet’: Lindt

So we have the brands and the types of chocolate, and the assignation. This is the use of dictionaries:

{'Dark': ‘Valor’, ‘Milk’: ‘Nestle’, ‘Semi sweet’: ‘Lindt’}

We have the keys and the values  and the keys need to be unique.

# Dicts

```
Chocolates_and_brands = {'Dark': ‘Valor’, ‘Milk’: Nestle, ‘Semi sweet’: Lindt}

> Chocolates_and_brands.keys()
> Chocolates_and_brands.values()
> Chocolates_and_brands[‘Dark’]

> Chocolates_and_brands[‘Nuts’] = ‘Nestle’
> del Chocolates_and_brands[‘Nuts’]
```

# Iterating through a dict

```
> for i in chocolates_and_brands.keys():
>         print(i)

> for chocolate_type in chocolates_and_brands.keys():
>         print(chocolate_type)
```

To do the same but printing the brands, what should I do?

```
 for chocolate_brand in chocolates_and_brands.values():
>         print(chocolate_brand)
```

# Iterating through a dict

We can also have both:

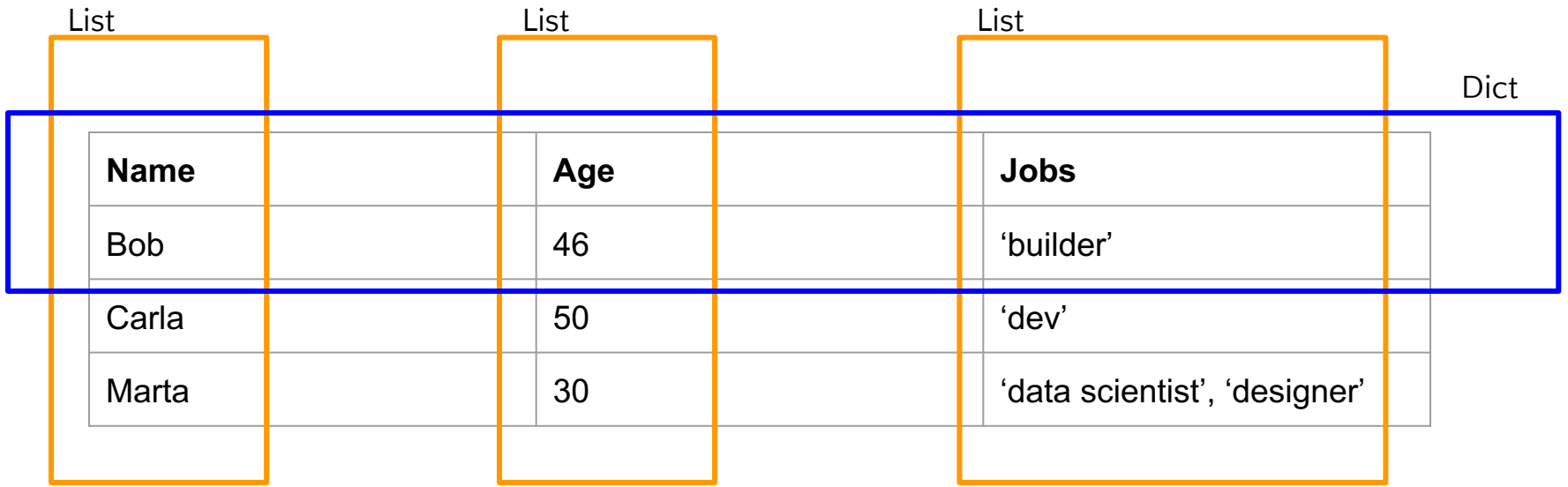
```
> for choc_type, chocolate_brand in chocolates_and_brands.items():
>         print(‘The best brand for the’ + choc_type + ‘is’
+chocolate_brand)
```

# Dicts

Values can be also lists.

```
Brands = {‘Nestle’: [‘dark’, ´milk´], ‘Valor’: []}
```

Or other dicts


List
List
List
Dict
Name
Age
Jobs
Bob
46
'builder'
Carla
50
'dev'
Marta
30
'data scientist', 'designer'

# Tuples - Definition

- A **tuple** is a sequence of arbitrary Python objects. In a tuple, items are separated by commas.
- They are used everywhere in Python, because they allow for patterns that are hard to reproduce in other languages.

What if you don't want to change the data in a list? Tuples are meant for that

```
> chocolates = ('dark', 'milk', 'semi sweet')
> chocolates[0]
```

```
> chocolates[1] = 'caramel filled' What happens?
```

```
>>> chocolates = ('dark', 'milk', 'semi sweet')
>>> chocolates[0]
'dark'
>>> chocolates[1] = 'c'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>>
```

# Sets - Definition

- A **set** contains an unordered collection of unique and immutable objects.
- Sets unlike lists or tuples can't have multiple occurrences of the same element

```
>>> small_primes = set()  # empty set
>>> small_primes.add(2)  # adding one element at a time
>>> small_primes.add(3)
>>> small_primes.add(5)
>>> small_primes
{2, 3, 5}
```

# Sets - Example

Unordered, unique, mutable. Like lists but **unique**.

```
> set()
> set(['Student1', 'Student2', 'Student3', 'Student4'])
```

What if there are 2 students with the same name? Then set is not our structure!

With sets you can do algebra sets. Union, intersection, subtraction…

https://docs.python.org/3.7/library/stdtypes.html#set-types-set-frozenset

# Sets - Example

```
>>> small_primes.add(1)  # Look what I've done, 1 is not a prime!
>>> small_primes
{1, 2, 3, 5}
>>> small_primes.remove(1)  # so let's remove it
>>> 3 in small_primes  # membership test
True
>>> 4 in small_primes
False
>>> 4 not in small_primes  # negated membership test
True
>>> small_primes.add(3)  # trying to add 3 again
>>> small_primes
{2, 3, 5}  # no change, duplication is not allowed
>>> bigger_primes = set([5, 7, 11, 13])  # faster creation
>>> small_primes | bigger_primes  # union operator `|`
{2, 3, 5, 7, 11, 13}
>>> small_primes & bigger_primes  # intersection operator `&`
{5}
>>> small_primes - bigger_primes  # difference operator `-`
{2, 3}
```