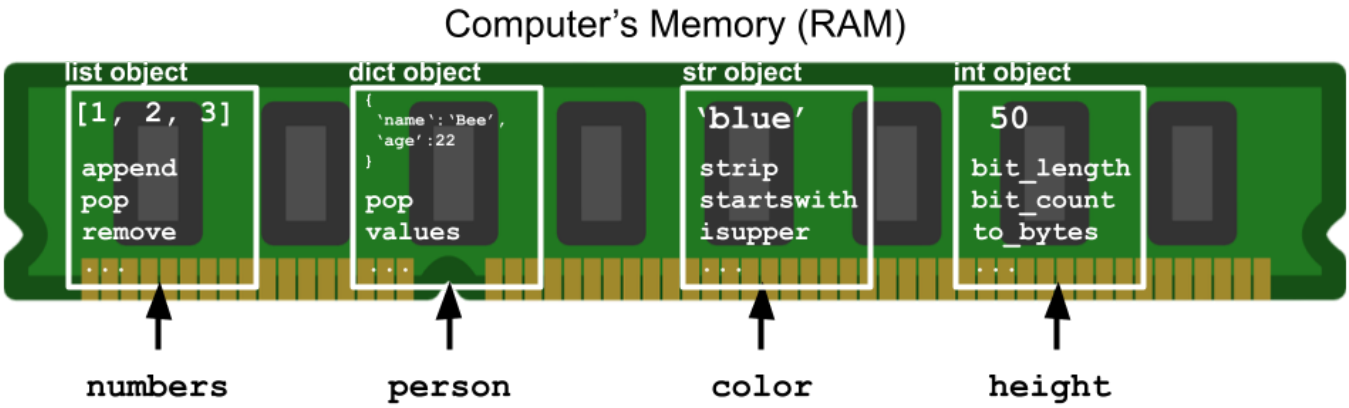
# Mad Methods

Recall...

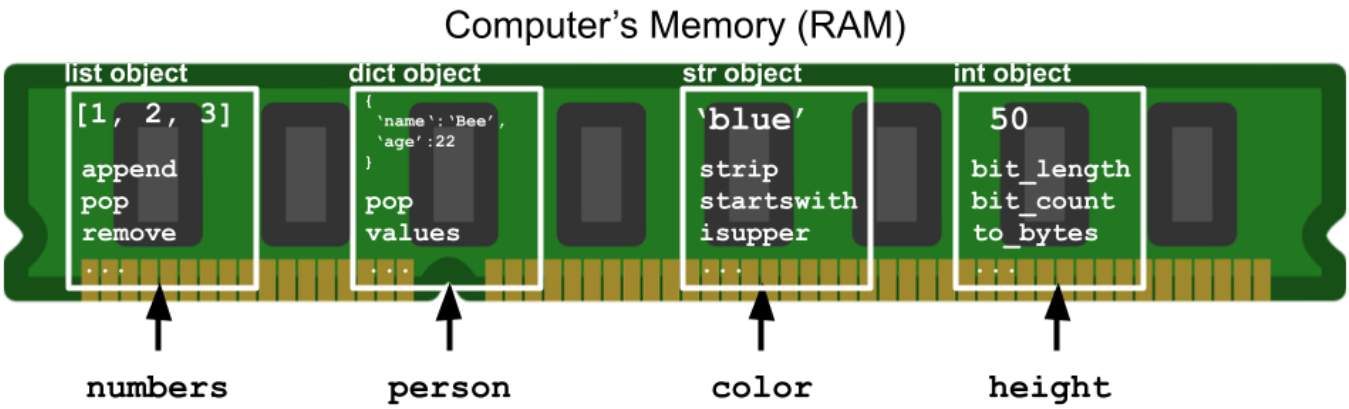
```
numbers = [1, 2, 3]
person = {
    'name': 'Bee',
    'age': 22
}
color = 'blue'
height = 50
```




Objects contain:

1. Value(s)
2. Things we can do to (or with) the value(s)--**methods**

Both of these depend on the object's type.




```
numbers = [1, 2, 3]
print(f'Initial: {numbers}')
```

```
numbers.append(4)
print(f'.append(4): {numbers}')
numbers.pop(0)
print(f'.pop(0): {numbers}')
numbers.remove(2)
print(f'.remove(2): {numbers}')
```

Methods are a subset of functions.

Methods:

```
numbers = [1, 2, 3]
person = {
    'name': 'Bee',
    'age': 22
}
color = 'blue'

numbers.append(4)
person.values()
color.upper()
```

Functions:

```
input('Enter a number: ')
print('Hello, world!')
type(2)
```

**What is one similarity and one difference between methods and functions?**

## Key similarities

- Called using parentheses
- Do something when they are called
- Can take inputs
- Can give outputs

## Key difference

Methods *belong* to an object and are always called *on* an object using dot notation.

`color.upper()`: The `upper` method is being called *on* `color`, which is a `str` object.

When we call methods *on* an object, the method does something *to* or *with* the value(s) of that object.

- `numbers.append(4)`: Adds the item `4` to the `numbers` list
- `person.values()`: Returns all values in the `person` dictionary
- `color.upper()`: Returns the `color` string in all uppercase letters

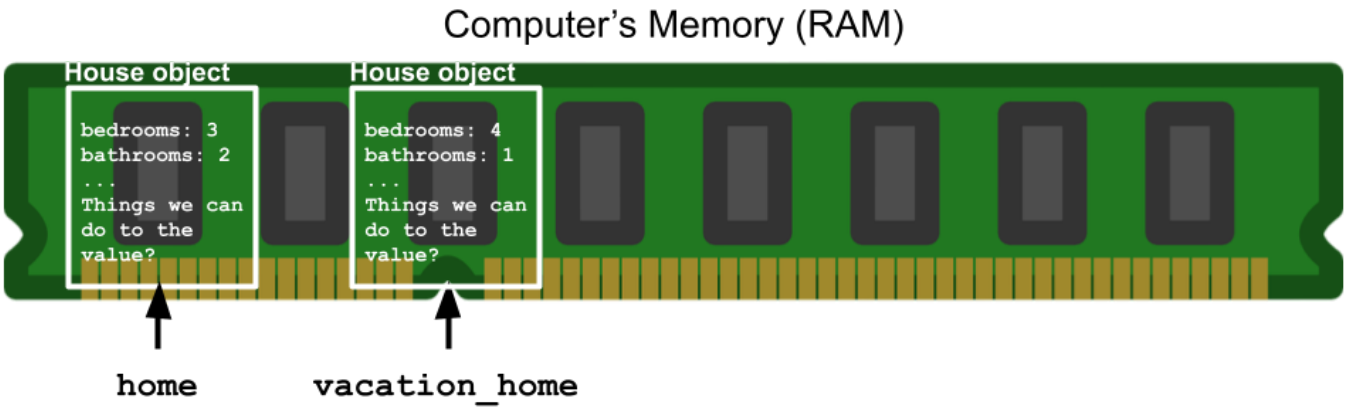
**Using the Internet, describe two other methods you can call on any object of type `str`. Write down the search query you used and the link to the resource you found.**

## Review

Last week, we discussed how to define our own type and how to add attributes (i.e. values) that every object of that type would have.

```
class House:
    """Represent a house."""
    def __init__(self, bedrooms, bathrooms):
        self.bedrooms = bedrooms
        self.bathrooms = bathrooms

home = House(3, 2)
print(home.bedrooms)
print(home.bathrooms)
vacation_home = House(4, 1)
print(vacation_home.bedrooms)
print(vacation_home.bathrooms)
```




**Can we access `home.color`? Why or why not?**

Nope--we never assigned the `self.color` attribute in the constructor method.

**Update the `__init__` method such that all objects of type `House` will have the attribute `color`, whose value will be passed as an argument to the constructor. Then, create a `House` object named `home`, passing in all of the required arguments.**

```
class House:
    """Represent a house."""
    def __init__(self, bedrooms, bathrooms, color):
        self.bedrooms = bedrooms
        self.bathrooms = bathrooms
        self.color = color

home = House(3, 2, 'red')
```

In our class definitions, we can also add **methods** that do something *to* or *with* the values of the object.

Values of `House` objects:

- Bedrooms
- Bathrooms
- Color

Things we could do to these values:

- Increment number of bedrooms ("add a bedroom")
- Increment number of bathrooms ("add a bathroom")
- Change the color ("paint")

Something we could do with these values:

- Generate a description of the house

Remember that values and methods of objects often reflect the real world!

We already know how to add methods!

```
class House:
    """Represent a house."""
    def __init__(self, bedrooms, bathrooms, color):
        self.bedrooms = bedrooms
        self.bathrooms = bathrooms
        self.color = color

    def __str__(self):
        return r"""
          _
       _|=|_________
      /             \
     /               \
```

```
  /__________________\
   ||  || /--\ ||  ||
   ||[]|| | .| ||[]||
 ()||__||_|__|_||__||()
( )|-|-|-|====|-|-|-|( )
^^^^^^^^^^^====^^^^^^^^^^^
        """

home = House(3, 2, 'red')
print(home)
```

**ELI5: What does the `__str__` method do?**

`__str__` should return the *string representation* of an object.

Whenever `House` objects are converted to a string (e.g. when passed into `print`), the string will be whatever is returned by `__str__`.

```
class House:
    """Represent a house."""
    def __init__(self, bedrooms, bathrooms):
        self.bedrooms = bedrooms
        self.bathrooms = bathrooms

home = House(3, 2)
```

To add a method to a class definition:

1. Indent under the class definition header.
2. Write the keyword `def`, then the name of the method, then parentheses that contain parameters--just like functions!
3. Start parameters with the `self` parameter, which stores a reference to the current object (i.e. the object the method is called on), followed by other parameters.
4. Use `self` to access attributes of the current object.

```
class House:
    """Represent a house."""
    def __init__(self, bedrooms, bathrooms, color):
        self.bedrooms = bedrooms
        self.bathrooms = bathrooms
        self.color = color

    def add_bedroom(self):
        self.bedrooms += 1

home = House(3, 2, 'red')
print(home.bedrooms)
home.add_bedroom()
print(home.bedrooms)
```

How do we add a method that takes an argument?

```
home.paint('white')
```

**Add a `paint` method that takes a color as an argument and updates the `color` attribute of the current object.**

```
class House:
    """Represent a house."""
    def __init__(self, bedrooms, bathrooms, color):
        self.bedrooms = bedrooms
        self.bathrooms = bathrooms
        self.color = color

    def add_bedroom(self):
        self.bedrooms += 1

    def paint(self, new_color):
        self.color = new_color

home = House(3, 2, 'red')
print(home.color)
home.paint('white')
print(home.color)
```

**Not using `House`**

```
home_bedrooms = 3
home_bathrooms = 2
home_color = 'red'

vacation_home_bedrooms = 4
vacation_home_bathrooms = 1
vacation_home_color = 'blue'

new_home_color = 'white'
if new_home_color in ['red', 'blue', 'white', 'green']:
    home_color = new_home_color
else:
    raise Exception('Invalid color.')

new_vacation_home_color = 'green'
if new_vacation_home_color in ['red', 'blue', 'white', 'green']:
    vacation_home_color = new_vacation_home_color
else:
    raise Exception('Invalid color.')

print(f'{home_color} | {home_bedrooms} br/{home_bathrooms} ba')
print(f'{vacation_home_color} | {vacation_home_bedrooms} br/{vacation_home_bathrooms} ba')
```

**Using `House`**

```
class House:
    """Represent a house."""
    def __init__(self, bedrooms, bathrooms, color):
        self.bedrooms = bedrooms
        self.bathrooms = bathrooms
        self.color = color

    def paint(self, new_color):
        if new_color in ['red', 'blue', 'white', 'green']:
            self.color = new_color
        else:
            raise Exception('Invalid color.')

    def __str__(self):
        return f'{self.color} | {self.bedrooms} br/{self.bathrooms} ba'

home = House(3, 2, 'red')
vacation_home = House(4, 1, 'blue')
home.paint('white')
vacation_home.paint('green')
print(home)
print(vacation_home)
```

**What is one advantage of using a `House` type to represent houses in our program?**

## Advantages of using classes

1. They organize and enforce relationships between values

```
home_bedrooms = 3
home_bathrooms = 2
```

```
home_color = 'red'
print(home_bedrooms)
print(home_bathrooms)
print(home_color)
```

v.s.

```
home = House(3, 2, 'red')
print(home.bedrooms)
print(home.bathrooms)
print(home.color)
```

2. Methods give descriptions to actions on values

```
home_color = 'red'
home_color = 'white' # What real world action does this reflect?
```

v.s.

```
home = House(3, 2, 'red')
home.paint('white') # Ah, we are "painting" the house
```

3. Methods allow us to repeat logic for every action

```
new_home_color = 'white'
if new_home_color in ['red', 'blue', 'white', 'green']:
    home_color = new_home_color
else:
    raise Exception('Invalid color.')
new_vacation_home_color = 'green'
if new_vacation_home_color in ['red', 'blue', 'white', 'green']:
    vacation_home_color = new_vacation_home_color
else:
    raise Exception('Invalid color.')
```

v.s.

```
def paint(self, new_color):
    if new_color in ['red', 'blue', 'white', 'green']:
        self.color = new_color
    else:
        raise Exception('Invalid color.')
```

```
home.paint('white')
vacation_home.paint('green')
```

Let's take our method skills to the next level...