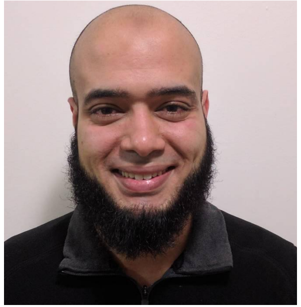
Data Structures

Built-in Hash Method

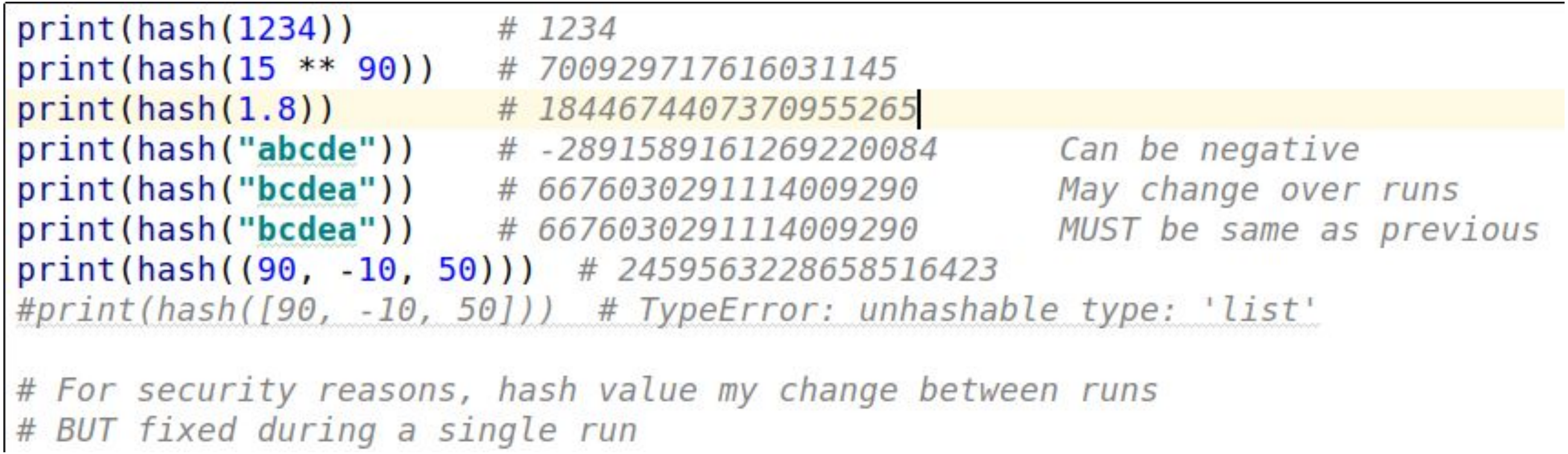
Mostafa S. Ibrahim
Teaching, Training and Coaching for more than a decade!

Artificial Intelligence & Computer Vision Researcher
PhD from Simon Fraser University - Canada
Bachelor / Msc from Cairo University - Egypt
Ex-(Software Engineer / ICPC World Finalist)



Built-in Hash Method

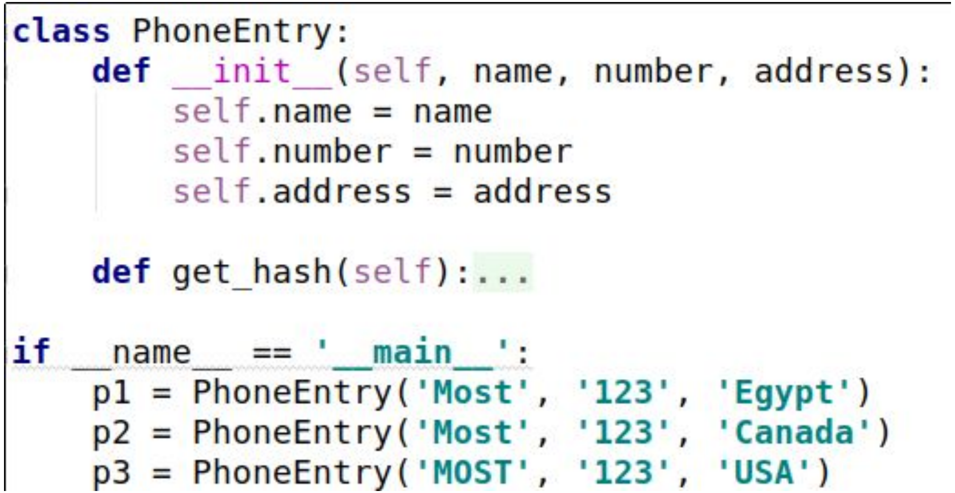
- We usually use **built-in** hash method
- Don’t apply on **mutable objects** such as lists
    - You can guess the reason after the end of this section



```
print(hash(1234))       # 1234
print(hash(15 ** 90))   # 70092971616031145
print(hash(1.8))        # 1844674407370955265
print(hash("abcde"))    # -2891589161269220084      Can be negative
print(hash("bcdea"))    # 6676030291114009290       May change over runs
print(hash("bcdea"))    # 6676030291114009290       MUST be same as previous
print(hash((90, -10, 50)))  # 2459563228658516423
#print(hash([90, -10, 50]))  # TypeError: unhashable type: 'list'

# For security reasons, hash value my change between runs
# BUT fixed during a single run
```

Smartphone Contacts Application

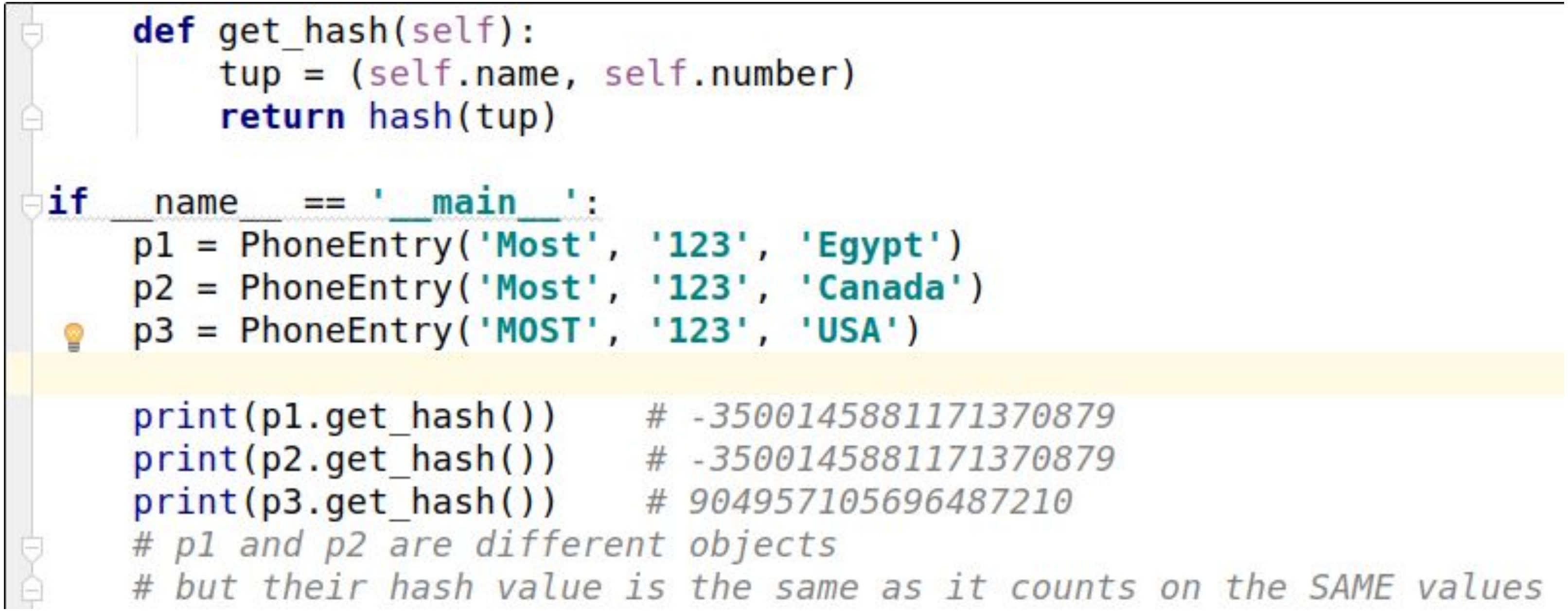
- Assume we have a class for phone entries consisting of 3 attributes
  - Name, Phone number and address
- How can we build a hash function for it?
  - First, determine what is really representative for an object?
  - Is it name only? name and number? all of them?



```python
class PhoneEntry:
    def __init__(self, name, number, address):
        self.name = name
        self.number = number
        self.address = address

    def get_hash(self):...

if __name__ == '__main__':
    p1 = PhoneEntry('Most', '123', 'Egypt')
    p2 = PhoneEntry('Most', '123', 'Canada')
    p3 = PhoneEntry('MOST', '123', 'USA')
```

Hashing the class

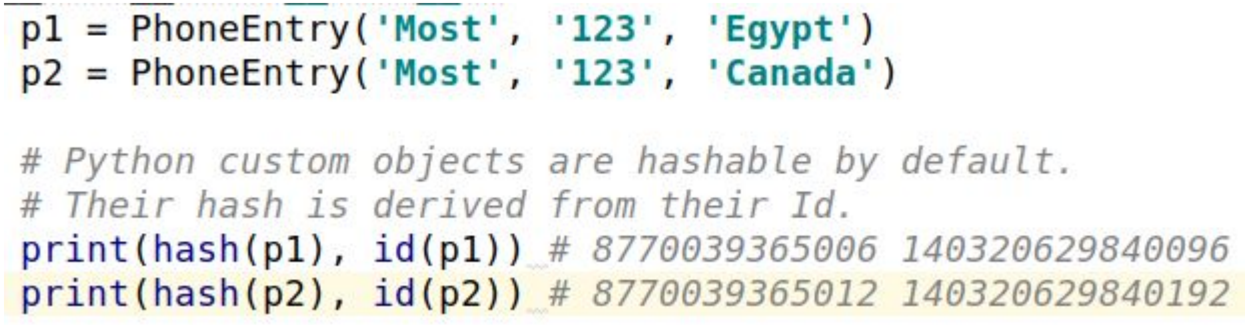
- Assume we picked ONLY name and number as representative
    - Just create a **tuple** of the entries and hash it



```
    def get_hash(self):
        tup = (self.name, self.number)
        return hash(tup)

if __name__ == '__main__':
    p1 = PhoneEntry('Most', '123', 'Egypt')
    p2 = PhoneEntry('Most', '123', 'Canada')
    p3 = PhoneEntry('MOST', '123', 'USA')

    print(p1.get_hash())    # -3500145881171370879
    print(p2.get_hash())    # -3500145881171370879
    print(p3.get_hash())    # 904957105696487210
    # p1 and p2 are different objects
    # but their hash value is the same as it counts on the SAME values
```

Next

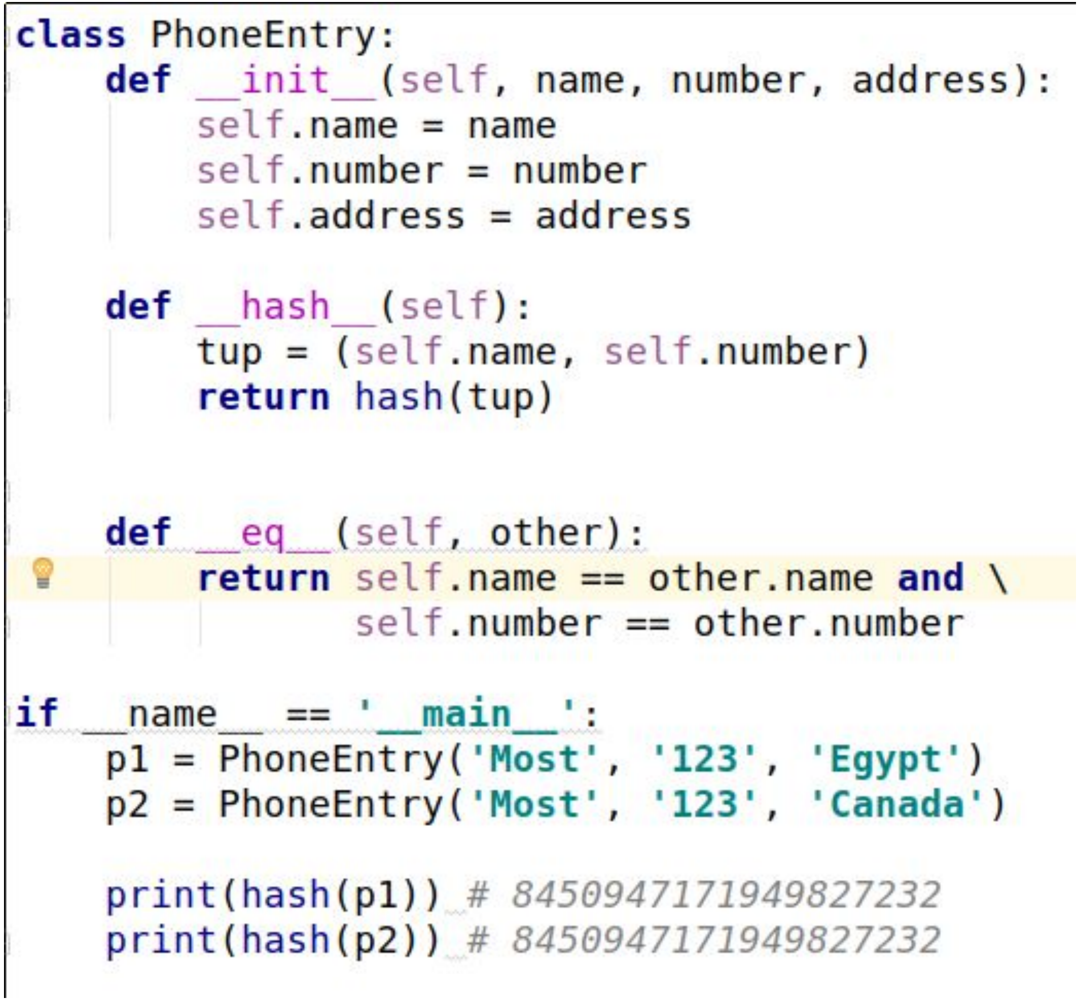
- I want to add here **programming/oop** information about built-in hashing
  - This is **plus** from me. If you don't get ~50% of that, it is ok
- Goal: we want to be able to do hash(p1) instead of p1.get_hash
  - This way, other classes can get the hash values without knowing our method
  - Below the same data in p1/p2 has different hash values!



```
p1 = PhoneEntry('Most', '123', 'Egypt')
p2 = PhoneEntry('Most', '123', 'Canada')

# Python custom objects are hashable by default.
# Their hash is derived from their Id.
print(hash(p1), id(p1)) # 8770039365006 140320629840096
print(hash(p2), id(p2)) # 8770039365012 140320629840192
```

Custom class

- We add BOTH **__hash__** and **__eq__** methods



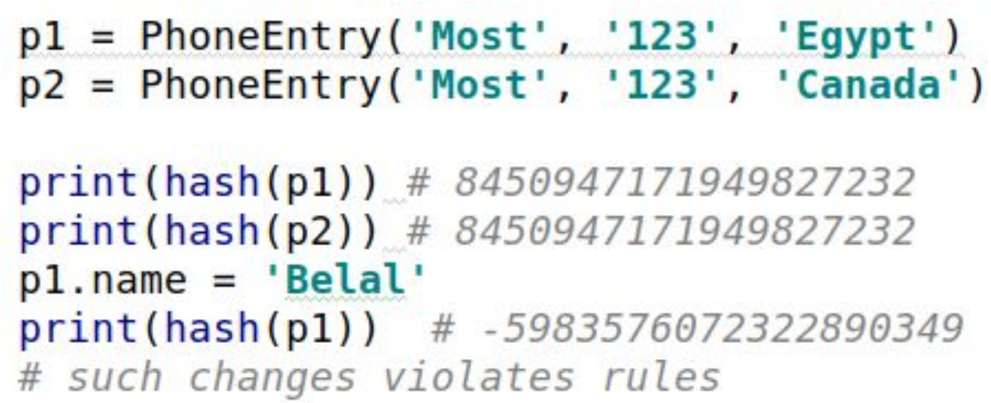
```
class PhoneEntry:
    def __init__(self, name, number, address):
        self.name = name
        self.number = number
        self.address = address

    def __hash__(self):
        tup = (self.name, self.number)
        return hash(tup)


    def __eq__(self, other):
        return self.name == other.name and \
               self.number == other.number

if __name__ == '__main__':
    p1 = PhoneEntry('Most', '123', 'Egypt')
    p2 = PhoneEntry('Most', '123', 'Canada')

    print(hash(p1)) # 8450947171949827232
    print(hash(p2)) # 8450947171949827232
```

Strict Rules

- For **correct** behaviour, you MUST implement **both** hash and eq methods
  - Never use __hash__ without __eq__
  - You can have __eq__, but never use the object with something that requires __hash__
    - You will get an error: TypeError: **unhashable** type
- Hash must return integer and Eq should return boolean
- If 2 objects are equal, they must have the SAME hash value
  - But the opposite is not true.
  - Different objects may have the same hash (collision)
- The hash value of an object MUST be the same during running
  - Hence, NEVER use hash for a mutable object like a list, as list content can change

Violating rules

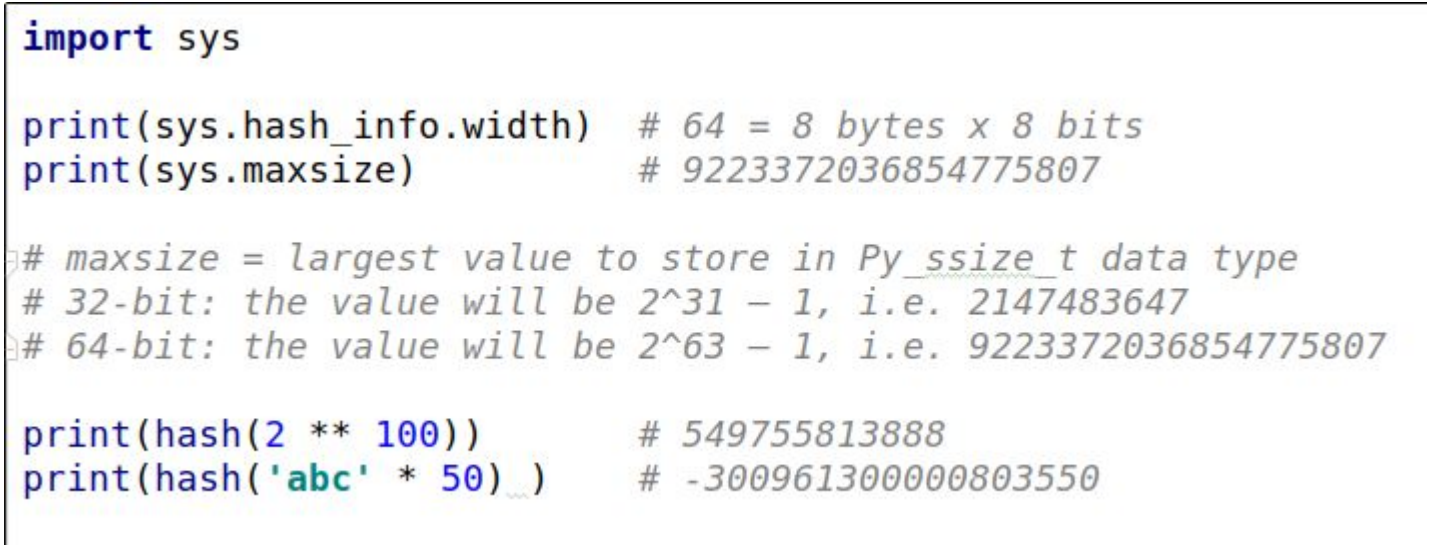
- Follow the rules strictly as the hash value is critical for the dictionary
- For future: there is @dataclass *decorator* that can freeze values



```
p1 = PhoneEntry('Most', '123', 'Egypt')
p2 = PhoneEntry('Most', '123', 'Canada')

print(hash(p1)) # 8450947171949827232
print(hash(p2)) # 8450947171949827232
p1.name = 'Belal'
print(hash(p1))  # -5983576072322890349
# such changes violates rules
```

Behind the scenes: Truncation

- hash() **truncates** the value returned from an object’s custom __hash__() method to the size of a **Py_ssize_t**.
    - 8 bytes on 64-bit builds and 4 bytes on 32-bit builds.



```
import sys

print(sys.hash_info.width)  # 64 = 8 bytes x 8 bits
print(sys.maxsize)          # 9223372036854775807

# maxsize = largest value to store in Py_ssize_t data type
# 32-bit: the value will be 2^31 – 1, i.e. 2147483647
# 64-bit: the value will be 2^63 – 1, i.e. 9223372036854775807

print(hash(2 ** 100))       # 549755813888
print(hash('abc' * 50) )    # -3009613000000803550
```

Behind the scenes: Negative values

- The hash function can return negative values
  - Not sure of technical reasons
  - I guess comes from their truncation / bits processing
  - I know in C due to **arithmetic overflow**
    - Recall hash_string2: when we keep multiply/add which overflows C integer limits
- Assume we want to force hash value to be in range [0, MAX-1]
  - Maybe we can make use of the maxsize (i.e. hash + maxsize + 1) to force it positive value
  - But we can just use hash % MAX to return a positive value in this range
    - Python's (%) always return a number having the **same sign as the denominator**
- For future: hashlib — Secure hashes and message digests

"Acquire knowledge and impart it to the people."

"Seek knowledge from the Cradle to the Grave."