Dicionários

Conhecidos também como HashMaps ou Arrays Associativos os dicionários são um misto entre o `set` e `list` e com certeza a estutura de dados mais importante da linguagem.

Curiosidade: Todos as tipos de dados do Python são implementados usando dicionários, os objetos possuem um método especial `__dict__`, experimente no terminal imprimir `int.__dict__`, até mesmo a resolução de nomes do Python é feita usando um dicionário `__builtins__.__dict__` quando digitamos `print` o Python vai ali naquele dicionário verificar se existe uma função chamada `print` lá dentro.

Vamos entender para que servem os dicionários com este exemplo usando tipos primários.

```
produto_nome = "Caneta"
produto_cor1 = "azul"
produto_cor2 = "branco"
produto_preco = 3.23
produto_dimensao_altura = 12.1
produto_dimensao_largura = 0.8
produto_em_estoque = True
produto_codigo = 45678
produto_codebar = None
```

Acima estamos representando um único produto em um programa de vendas, é uma única caneta porém precisamos de 9 objetos de diferentes tipos.

E para resolver este problema que temos os tipos compostos, eles são `containeres`, objetos que podem agrupar mais de um tipo e mais de uma unidade de informação dentro deles.

Dicionários são criados com `{ }` ou através da classe `dict()`, é bom ter cuidado para não os confundir com sets já que sets também usam `{ }` e a diferença principal é o fato de que no set cada posição armazena apenas um valor e nos dicionários podemos colocar 2 valores em cada posição.

Um desses valores é chamado de chave `key` e o outro valor `val` e são separados por `:`, veja com um dicionário a mesma informação pode ser representada com:

```
produto = {
        "nome": "Caneta",
        "cor1": "azul",
        "cor2": "branco",
        "preco": 3.23,
        "dimensao_altura": 12.1,
        "dimensao_largura":  0.8,
        "em_estoque": True,
        "codigo": 45678,
        "codebar": None,
}
```

Agora temos um único objeto `produto` do tipo `dict` e isso torna nosso programa muito melhor organizado, facilita operações e deixa a complexidade menor também pois os dicionários assim como os sets também implementam a Hash Table, ou seja, as operação de acesso são `O(1)` super rápidas.

Assim como as listas os dicionários podem receber subscrições a partir de uma chave, ou seja, usamos `[]` e dentro passamos a `key` que queremos acessar.

```
>>> produto["nome"]
'Caneta'
>> produto["codigo"]
45678
```

Além disso podemos diminuir a redundancia colocando objetos compostos dentro do dicinário, ao invés de 2 chaves para representar cor podemos criar uma lista de cores e ao invés de 2 chaves para dimensoes podemos ter um subdicionário.

```
produto = {
        "nome": "Caneta",
        "cores": ["azul", "branco"],
        "preco": 3.23,
        "dimensao" {
        "altura": 12.1,
            "largura":  0.8,
    },
        "em_estoque: True,
        "codigo": 45678,
        "codebar": None,
}
```

As chaves de um dicionário podem ser de qualquer tipo de dados que seja compatível com hash table, exemplo:

```
d = {
    10: "Bruno",
    True: 45,
    False: None,
    None: True,
    (4,5,6): "uma tupla",
}

>>> d[(4, 5, 6)]
'uma tupla'
```

Mas não podemos usar tipos unhashable.

```
d = {[1, 2, 3]: "uma lista"}
...
TypeError: unhashable type: 'list'
```

Já os valores podem ser de qualquer tipo sem restrições!

Sintaxe

Podemos iniciar um dicionário vazio e depois ir adicionando elementos dentro dele.

```
cliente = {}
# ou
cliente = dict()
```

CRUD

E usar as operações de CRUD (Create, Read, Update, Delete)

Criar - Adicionar chave+valor

```
cliente["nome"] = "Bruno"
```

Ler valor a partir de uma chave

```
>>> cliente["nome"]
'Bruno'
```

Update - Alterar valor a partir de uma chave

```
cliente["nome"] = "Bruno Rocha"
```

Delete - Remover um valor e sua chave.

```
del cliente["nome"]
```

> A keyword `del` remove uma variável que foi atribuida e pode ser usada com chaves de dicionários.

Buscas

O dicionário implementa Hash Table, ele também é conhecido como **hash map** e portanto as buscas em dicionário quando feitas por chave tem acesso constante O(1).

```
"nome" in cliente
True
```

`in` invoca o protocolo `Lookupable` através do método `__contains__` e efetua a busca imediata sem necessidade de iterar todo o dicionário para encontrar uma chave, e assim como os `sets` as chaves não podem se repetir.

Métodos de Lookup

Obter uma lista de chaves

```
cliente = {"nome": "Bruno", "cidade": "Viana"}

cliente.keys()
dict_keys(["nome", "cidade"])
```

Obter uma lista de valores

```
cliente.values()
dict_keys(["Bruno", "Viana"])
```

Obter uma lista de tuplas contendo chave e valor

```
cliente.items()
dict_items([("nome", "Bruno"), ("cidade", "Viana")])
```

Iterando com `for`

```
# apenas chaves
for key in cliente:

```

```
    print(key)

nome
cidade


# Buscando os valores com subscrição
for key in cliente:
    print(key, "-->", cliente[key])

nome-->Bruno
cidade-->Viana


# Com desenpacotamento de tuplas
for key, value in cliente.items():
    print(key, "-->", value)

nome-->Bruno
cidade-->Viana
```

Combinando 2 dicionários

```
# informacao original
cliente = {"nome": "Bruno", "cidade": "Viana"}

# informacao adicional
extra = {"pais": "Portugal"}

# Informação final
final = {**cliente, **extra}
print(final)
{"nome": "Bruno", "cidade": "Viana", "pais": "Portugal"}
```

Ou fazendo `update` in place.

```
# informacao original
cliente = {"nome": "Bruno", "cidade": "Viana"}

# informacao adicional
extra = {"pais": "Portugal"}

# Cliente atualizado
cliente.update(extra)
print(cliente)
{"nome": "Bruno", "cidade": "Viana", "pais": "Portugal"}
```

Erros

Caso uma chave não exista no dicionário o Python estoura um erro chamado `KeyError`

```
print(cliente["telefone"])
...
```

```
KeyError 'telefone'
```

Para evitar o erro podemos usar o método `get` que busca a chave e caso não exista retorna um valor padrão que inicialmente é `None`

```
print(cliente.get("telefone"))
'None'

print(cliente.get("telefone", "191"))
'191'
```