OBXECTOS

EN

PYTHON

Evelyn Berezin (1925-2018)

- Inventora (1957) do primeiro ordenador de oficina, gardaba libros e contas e automatizaba un sistema bancario nacional
- Creadora (1971) do primeiro software procesador de texto
- Desenvolvedora (1962) do primeiro sistema software de reservas de pasaxeiros (60 cidades) para a liña aérea United Airlines, o sistema máis grande ata entón construído

Programación orientada a obxectos

- O paradigma da programación orientada a obxectos fusiona os datos e as funcións que operan sobre eses datos dentro dun novo tipo de dato chamado **CLASE**.
- A cada variable dunha clase chámaselle **OBXECTO**.
- Propiedades do paradigma orientado a obxectos: encapsulación, herdanza e polimorfismo.

Clases e obxectos

- As **clases** son modelos de cousas que serven para crear obxectos concretos.
- Unha clase contén propiedades (atributos) e métodos:
  - As **propiedades** son características intrínsecas do obxecto e represéntanse con variables.
  - Os **métodos** son accións que pode realizar o obxecto e represéntanse con funcións.
- O proceso de **crear un obxecto** chámaselle instanciar unha clase na que se lle da un valor concreto ás súas propiedades.

Clases e obxectos

- Definición dunha clase:

```
class nome_clase:
    # especificar propiedades (variable)
    declaración_propiedade1
    declaración_propiedade2
    # especificar métodos (funcións)
    def nome_metodo1(self, [parametros]):
        # definicion do método
    def nome_metodo2(self, [parametros]):
```

- Instanciación dunha clase (creación dun obxeto):

```
nome_obxeto=nome_clase()
```

- Acceder os atributos e métodos dun obxeto:

```
nome_obxeto.nome_propiedade
```

Clases e obxectos

- Onde **`self`**:
  - É o primeiro argumento de calquera método.
  - Fai referencia á propia clase e ao seu contido.
  - É un método implícito e nunca se pasa como parámetro cando se chama un método.
- Exemplo:

```
#!/usr/bin/python3
# definicion dunha clase
class minhaclase:
  i=3.45 # declaracion de propiedade
  def f(self):
    return "Ola a todos"

# creacion dun obxeto de tipo minhaclase
x = minhaclase()
# acceso as propiedades e metodos do obxeto
print(x.i)
print(x.f())
```

Clases e obxectos

- Na práctica o exemplo anterior e pouco útil porque non permite inicializar os atributos na creación do obxecto.
- O paso de argumentos na creación de obxectos realízase definindo o método implícito da clase **`__init__()`**:

```
__init__(self, [parametros]):
    # asignación de argumentos a propiedades da clase
```

- __init__() execútase de forma automática ó crear o obxecto (outras linguaxes de programación chámanlle construtor).
- Na definición desta función podes crear atributos da clase e pasarlle parámetros cos que inicializar estes atributos na creación do obxecto.
- **`__str__(self)`**: método chamado para obter unha cadea de texto que represente o obxeto.

Clases e obxectos

- Exemplo de creación de obxecto dunha clase dándolle valores aos seus atributos co método __init__():

```
#!/usr/bin/python3
# definicion dunha clase
class complexo:
  # definicion do metodo _init_()
  def __init__(self, partereal, parteimag):
    self.r = partereal # asignacion
    self.i = parteimag # asignacion
# creacion de obxetos de tipo complexo
x = complexo(3.0, 4.0)
z = complexo(2.5, -1.3)
# acceso as propiedades e metodos do obxeto
print("Complexo x= ",x.r, "+ j ", x.i)
print("Complexo z= ",z.r, "+ j ", z.i)
```

Clases e obxectos

- Os métodos poden utilizar os métodos da clase a través de **`self`**:

```
class Bolsa:
    def __init__(self):
        self.datos = []
    def agregar(self, x):
        self.datos.append(x)
    def dobleagregar(self, x):
        self.agregar(x)
        self.agregar(x)
```

Algúns obxectos en Python

- **`string`**: cadeas de caracteres
- **`list`**: obxecto de tipo lista
- **`File`**: obxecto para traballar con arquivos
- **`Dict`**: como as listas e tuplas, un diccionario (obxeto de tipo **`Dict`**) permite almacenar datos de distinto tipo, pero accédese ós elementos con claves no canto do índice.
- **`Exception`**: obxeto para xestionar os erros de execucción dos programas.

Métodos do obxecto *string*

Máis información en: http://librosweb.es/libro/python/capitulo_6.html e https://docs.python.org/2/library/stdtypes.html#string-methods

- De formato (todos os métodos devolven unha copia da cadea):
  - **`capitalize()`**: coa primeira letra en maiúscula.
  - **`lower()`**: a cadea en minúsculas.
  - **`upper()`**: a cadea en maiúsculas.
  - **`center(lonxitude[, "caracter de recheo"]):`** a cadea centrada.
  - **`ljust(lonxitude[, "caracter de recheo"])`**: a cadea aliñada a esquerda.
  - **`rjust(lonxitude[, "caracter de recheo"])`**: igual a anterior para a dereita.
  - **`zfill(lonxitude)`**: a cadea con ceros a esquerda ate completar lonxitude.
- De procura (todos os métodos devolven un enteiro):
  - **`count("subcadea" [, posicion_inicio, posicion_fin])`**: número de veces que aparece a subcadea na cadea.
  - **`find("subcadea" [, posicion_inicio, posicion_fin])`**: primeira posición que aparece a subcadea na cadea. Se non se atopa devolve -1.

Métodos do obxecto *string*

Máis información en: http://librosweb.es/libro/python/capitulo_6.html e https://docs.python.org/2/library/stdtypes.html#string-methods

- De formato (todos os métodos devolven unha copia da cadea):
  - **`capitalize()`**: coa primeira letra en maiúscula.
  - **`lower()`**: a cadea en minúsculas.
  - **`upper()`**: a cadea en maiúsculas.
  - **`center(lonxitude[, "caracter de recheo"]):`** a cadea centrada.
  - **`ljust(lonxitude[, "caracter de recheo"])`**: a cadea aliñada a esquerda.
  - **`rjust(lonxitude[, "caracter de recheo"])`**: igual a anterior para a dereita.
  - **`zfill(lonxitude)`**: a cadea con ceros a esquerda ate completar lonxitude.
- De procura (todos os métodos devolven un enteiro):
  - **`count("subcadea" [, posicion_inicio, posicion_fin])`**: número de veces que aparece a subcadea na cadea.
  - **`find("subcadea" [, posicion_inicio, posicion_fin])`**: primeira posición que aparece a subcadea na cadea. Se non se atopa devolve -1.

Métodos do obxecto *string*

- De validación (todos os métodos devolven **True** ou **False**):
  - **`startswith("subcadea" [, posicion_inicio, posicion_fin]): true se a cadea comeza pola subcadea dada.`**
  - **`endswith("subcadea" [, posicion_inicio, posicion_fin]):`** igual a anterior para final de cadea.
  - **`isalnum()`**: true se a cadea é alfanumérica.
  - **`isalpha()`**: true se a cadea é alfabética.
  - **`isdigit()`**: true se a cadea é numérica.
  - **`islower()`**: true se unha cadea contén só minúsculas.
  - **`isupper()`**: true se unha cadea contén so maiúsculas.
  - **`isspace()`**: true se unha cadea só contén espacios en branco.
- De substitución (devolven unha cadea de caracteres):
  - **`replace("subcadea a buscar", "subcadea para reemplazar")`**: a cadea co texto reemplazado.
  - **`strip(["caracter"])`**: elimina caracteres a esquerda e dereita da cadea.
  - **`lstrip(["caracter"])`**: elimina caracteres a esquerda da cadea.
  - **`rstrip(["caracter"])`**: elimina caracteres a dereita da cadea.

Métodos do obxecto *string*

- De unión e división:
  - **`join(iterable)`**: une a cadea a todos os elementos do iterable.
  - **`partition("separador")`**: parte unha cadea en tres partes utilizando un separador. Devolve unha tupla cos tres elemento (antes separador, separador, despois separador).
  - **`split("separador")`**: parte unha cadea en partes utilizando un separador. Devolve unha lista cos elementos separados por separador.
  - **`rsplit([sep[, maxsplit]]):`** devolve unha lista de palabras na cadea, usando sep como delimitador. Se non se especifica sep, o separador son espazos en branco.

Métodos do obxecto `list`

- **`append("novo_elemento")`**: engade un elemento o final da lista.
- **`extend(outra_lista)`**: engade outra_lista ó final da lista.
- **`insert(posicion, "novo_elemento")`**: engade un elemento en posicion.
- **`pop()`**: elimina o último elemento da lista.
- **`pop(indice)`**: elimina o elemento na posición índice na lista.
- **`remove("valor")`**: elimina os elementos na lista con valor.
- **`reverse()`**: invertir a orde nunha lista.
- **`sort()`**: ordear a lista en orde ascendente.
- **`sort(reverse=True)`**: ordear a lista de forma descendente.
- **`count(elemento)`**: contar o número de veces que a lista contén elemento.
- **`index(elemento[, indice_inicio, indice_fin]`**): devolve o número de índice de elemento.

Métodos do obxecto `File`

- **`f=open(ruta do arquivo, modo de apertura)`**: abre o arquivo especifica e devolve un obxeto de tipo **`File`** (f).
  - Modo de apertura pode ser:
    - r : lectura
    - w : escritura
    - a : engadir ó final do arquivo
    - Rb, wb e ab: igual que as antereriores pero para ler e escribir en modo binario.
    - r+, w+ e a+ : abrir para lectura e escritura

Métodos do obxecto `File`

Máis información en: https://docs.python.org/2/library/stdtypes.html#file-objects

- **`seek(byte[, whence])`**: move o punteiro ó byte indicado (por defecto, desde o inicio do arquivo). O argumento whence é opcional e pode ser (hai que importar o módulo os):
  - SEEK_SET ou 0 (desde o inicio)
  - SEEK_CUR ou 1 (desde a posición actual)
  - SEEK_END ou 2 (desde o final do arquivo).
- Exemplos:
  - from os import *
  - f.seek(2, SEEK_CUR) avanza dous desde a posición actual.
  - f.seek(-3, SEEK_END) a tres bytes do final do arquivo.

Métodos do obxecto `File`

Máis información en: https://docs.python.org/2/library/stdtypes.html#file-objects

- **`read([byte])`**: le o contido do arquivo. Se se lle pasa un tamaño, le só os bytes indicados.
- **`readline()`**: le unha liña do arquivo.
- **`readlines()`**: le todas as liñas do arquivo.
- **`tell()`**: devolve a posición actual do punteiro.
- **`write(cadea)`**: escribe cadea dentro do arquivo.
- **`writelines(secuencia)`**: secuencia será calquer iterable (lista, tupla, dict) cuxos elementos escribiranse cada un nunha liña.

Métodos do obxectos `File`

- **`f.close()`**: pecha o arquivo f.
- Propiedades do obxecto File:
  - **closed**: devolve True se o arquivo se pechou correctamente e se non devolve False.
  - **mode**: devolve o modo de apertura.
  - **name**: devolve o nome do arquivo.
  - **encoding**: devolve a codificación do arquivo de texto.

Obxecto `Dict`

- Creacción:
  - `meuDict ={'clave1':5, 'clave2':'ola', 'clave3': [2,5,7], 'clave4':True}`
  - `meuDict # saida {'clave1': 5, 'clave2': 'ola', 'clave3': [2, 5, 7], 'clave4': True}`
- Acceso e modificación:
  - `meuDict['clave2']='outra cadea'`
  - `meuDict # saida {'clave1': 5, 'clave2': 'outra cadea', 'clave3': [2, 5, 7], 'clave4': True}`
- Eliminación de elementos:
  - `del(meuDict['clave3'])`
  - `meuDict # saida {'clave1': 5, 'clave2': 'outra cadea', 'clave4': True}`

Métodos de obxetos `Dict`

- **`clear()`** : valeirar un diccionario.
- **`copy()`** : fai unha copia do diccionario.
- **`update(dict2)`**: concatena dous diccionarios.
- **`items()`**: devolve claves e valores do diccionario.
- **`keys()`**: devolve unha lista coas claves do diccionario.
- **`values()`**: devolve unha lista cos valores do diccionario.
- Función **`len(diccionario)`** devolve o número de elementos do diccionario.

Exemplo co obxeto `Dict`

```
meuDict ={'clave1':5, 'clave2':'ola', 'clave3':[2,5,7], 'clave4':True} #crea diccionario
print('Diccionario meuDict: ',meuDict) # visualiza contido de diccionario
meuDict['clave2']='outra cadea' # modifica diccionario
print('meudict despois modificacion: ',meuDict)
meuDict['clave5'] = "valor 5" # Engade unha nova entrada
print('meudict despois de engadir: ',meuDict)
del(meuDict['clave3']) # borra elemento de diccionario
print('meuDict despois borrar elemento: ',meuDict)
dict2 = meuDict.copy() # copia meuDict en dict2
print('dict2: ',dict2)
dict2.clear() # elimina os elementos de dict2
print('dict2 despois borrar elementos: ',dict2)
dict3={'color': 'azul', 'cantidad': 5} # crea diccionario dict3
meuDict.update(dict3) # concatena dict3 con meuDict
print('meuDict despois concatenar: ', meuDict)
del dict3 # borra o diccionario
print('Lista claves: ', meuDict.keys())
print('Lista valores: ', meuDict.values())
print('Existe clave6: ', 'clave6' in meuDict)
print('get valor clave 6: ', meuDict.get('clave6', 'non existe esta clave'))
print('Elementos na lista: ', len(meuDict))
for clave, valor in meuDict.items():
    print('O valor da clave %s e %s' % (clave, valor))
```

Excepcións

- As excepcións son erros detectados por Python durante a execución dun programa. Cando o programa atopa unha situación anómala xera ou lanza un evento (obxeto chamado excepción) informando que hai un problema.
- Posibles excepcións son: división por cero, arquivo que non existe, etc.
- Se a excepción non se captura, interrómpese o fluxo de control e o programa remata anticipadamente.
- Python utiliza a construcción **`try/except`** para capturar e tratar as excepcións. O bloque **`try`** (intentar) define o fragmento de código no que cremos que se produce unha excepción. O bloque **`except`** (excepción) permite indicar o que se fará se se produce dita excepción.
- Outras formas de manexar excepcións:
  - **`raise`**: dispara unha excepción manualmente no código.
  - **`assert`**: dispara unha excepción condicionalmente no código.

Excepcións incorporadas en Python

Todas as excepcións en: https://docs.python.org/2/library/exceptions.html

- **`Exceptions`**: clase raiz para todas as excepcións.
- **`AttributeError`**: dispárase cando falla unha referencia a atributo ou unha asignación.
- **`IOError`**: ocorre cando se intenta abrir un arquivo que non existe (entre outras cousas).
- **`IndexError`**: ocorre cando non existe un índice sobre unha secuencia.
- **`KeyError`**: disparada cando non existe unha chave sobre un mapping (diccionarios).
- **`NameError`**: disparado cando un nome ou variable non existe.
- **`SyntaxError`**: disparado cando o código esta mal formado.
- **`TypeError`**: dispárase cando unha operación ou función se aplica a un obxecto de tipo equivocado.
- **`ValueError`**: disparada cando unha operación ou función se aplica a un obxecto correcto pero con un valor inapropiado.
- **`ZeroDivisionError`**: disparado cando o segundo argumento dunha división ou operación de módulo é cero.
- **`NotImplementedError`**: Na definición de clases base, os métodos abstractos deberían lanzar esta excepción cando as clases derivadas teñan que redefinir ou sobrecargar o método.

Exemplos de manexo de excepcións

- Comprobar que un arquivo non existe cando se quere ler:

```
#!/usr/bin/python3
try:
  arquivo=open('datos.txt', 'r')
  for linha in arquivo:
    print(linha)
  arquivo.close()
except IOError:
  print('O arquivo datos.txt non existe')
```

- División por cero ou tipo de entrada incorrecto:

```
#!/usr/bin/python3
try:
  x = float(input('Introduce 1er numero: '))
  y = float(input('Introduce 2do numero: '))
  print(x/y)
except ZeroDivisionError:
   print("O segundo numero non debe ser cero.")
except ValueError:
  print("Algun operando non e un numero")
```