Introducción a la Computación Científica con Python

Mas programación en Python

Diego Passarella

Víctor Viana

EDUCACION PERMANENTE
Universidad de la República

SEDE TACUAREMBÓ
CENUR NORESTE

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

Contenido

- Variables, tipos de datos, asignaciones
- Operadores y comparaciones
- Estructuras de control: condicionales, ciclos, funciones
- Tipos de datos compuestos: cadenas, listas, diccionarios
- Ejemplo y Ejercicios

Lectura, asignación y despliegue

```
# programa que calcula el área de un triángulo

# ingreso de datos
altura = input('altura: ')
base   = input('base: ')

# cálculo del área
area = base * altura / 2

# mostrar resultado
print u"El área del triángulo es ", area,
```

Identificadores en Python

- Usados para dar nombre a los diferentes objetos que componen un programa:
  - variables
  - funciones
  - clases
  - módulos
  - paquetes
- En el ejemplo anterior: altura, base, input

Sintaxis de los identificadores

- Definición: Un **identificador** es una secuencia de caracteres **alfanuméricos**, el primero de los cuáles debe ser **alfabético**.
- Gramática BNF (Backus/Naur Form[1]):

```
identifier ::=  (letter|"_") (letter | digit | "_")*
letter     ::=  lowercase | uppercase
lowercase  ::=  "a"..."z"
uppercase  ::=  "A"..."Z"
digit      ::=  "0"..."9"
```

[1] https://es.wikipedia.org/wiki/Notación_de_Backus-Naur

Mayúsculas y Minúsculas

- Python es un lenguaje **case sensitivo**: distingue mayúsculas de minúsculas
- Así por ejemplo, los identificadores casa, CASA, Casa y cAsA son todos distintos.
- Lenguajes **case insensitivos**: Pascal, Basic, Fortran
- Lenguajes **case sensitivos**: C, Java, Perl, Python

Palabras reservadas

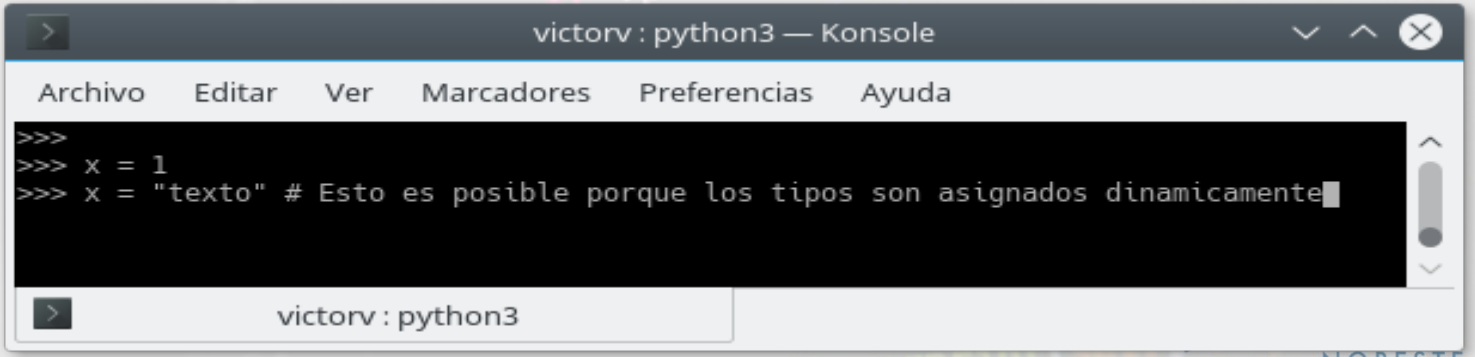
- También llamadas **keywords**
- Son ciertos identificadores cuyo uso se prohíbe
- Los utiliza el lenguaje con propósitos particulares como parte de las instrucciones
- En el ejemplo anterior se utiliza la palabra reservada **print**,
- Muchos editores colorean las palabras reservadas (emacs, vim, gedit, kate, idle)

Palabras reservadas en Python

```
and       del       from      not       while
as        elif      global    or        with
assert    else      if        pass      yield
break     except    import    print
class     exec      in        raise
continue  finally   is        return
def       for       lambda    try
```

Variables

- Se definen de forma dinámica: No se tiene que especificar cuál es su tipo de antemano.
- Puede tomar distintos valores en otro momento, incluso de un tipo diferente al que tenía previamente.
- Se usa el símbolo = para asignar valores.



Memoria

- El programa anterior utilizará tres variables: altura, base, area
- Una variable está formada por:
  - un nombre (identificador)
  - un valor
  - un tipo
- El valor y el tipo de una variable puede cambiar en el transcurso de un programa
- La memoria de un programa está constituida por el conjunto de variables utilizadas
- Más adelante: Espacio de nombres

Instrucción de Asignación

- El valor de una variable puede ser modificado en el transcurso del programa.
- La instrucción de asignación tiene esta forma:
  - identificador = expresión
- Asigna el valor a la variable correspondiente
- Si ya tenía valor, se pierde (asignación **destructiva**)
- Si no tenía valor, la variable se **crea** con la asignación (creación dinámica)
- El **valor** puede ser una expresión a ser evaluada

Instrucciones y Programas

- Un programa es una secuencia de instrucciones
- Dos tipos de instrucciones:
  - simples: expresión, asignación, break, continue y otras
  - compuestas: if, while, for y otras@

Ejemplos de asignaciones

```
altura = 0

base = 4.3 / 2.5

y = x * 1

cadena = "hola que tal"


contador = contador + 1
```

Sintaxis de la Asignación

```
assignment_stmt ::=  (target_list "=")+ (expression_list | yield_expression)
target_list     ::=  target ("," target)* [","]
target          ::=  identifier
                     | "(" target_list ")"
                     | "[" target_list "]"
                     | attributeref
                     | subscription
                     | slicing
```

Asignación en cadena

Se puede asignar el mismo valor a muchas variables:

```
a = b = c = d = 1
```

es equivalente a

```
a = 1
b = 1
c = 1
d = 1
```

Asignación en Paralelo

Se pueden realizar varias asignaciones en paralelo:

```
a,b = 0,1
```

Asigna el valor 0 a `a` y el valor 1 a `b`

Es útil para intercambiar el valor de dos variables:

```
a,b = b,a
```

Notar que no es lo mismo que hacer:

```
a = b
b = a
```

El Concepto de Tipo

- Los tipos permiten indicar la característica de los valores (datos) manipulados en un programa.
- Toda variable tiene asociado un tipo.
- Esto ocurre también con los **operadores** y las **funciones**.
- Poseer tipos permite detectar ciertos errores de construcción en el código (chequeo de tipos).
- Por ejemplo: 3 + 4 es correcto (integer)
- 3 / 'a' es incorrecto
- En Python el chequeo de tipos se hace en tiempo de ejecución (tipado **dinámico**)

Tipos de datos

Tipo	Clase	Notas	Ejemplo
`str`	Cadena	Inmutable	`'Cadena'`
`unicode`	Cadena	Versión Unicode de `str`	`u'Cadena'`
`list`	Secuencia	Mutable, puede contener objetos de diversos tipos	`[4.0, 'Cadena', True]`
`tuple`	Secuencia	Inmutable, puede contener objetos de diversos tipos	`(4.0, 'Cadena', True)`
`set`	Conjunto	Mutable, sin orden, no contiene duplicados	`set([4.0, 'Cadena', True])`
`frozenset`	Conjunto	Inmutable, sin orden, no contiene duplicados	`frozenset([4.0, 'Cadena', True])`
`dict`	Mapping	Grupo de pares clave:valor	`{'key1': 1.0, 'key2': False}`
`int`	Número entero	Precisión fija, convertido en *long* en caso de overflow.	`42`
`long`	Número entero	Precisión arbitraria	`42L` ó `456966786151987643L`
`float`	Número decimal	Coma flotante de doble precisión	`3.1415927`
`complex`	Número complejo	Parte real y parte imaginaria *j*.	`(4.5 + 3j)`
`bool`	Booleano	Valor booleano verdadero o falso	`True o False`

Tipos Numéricos

- Números enteros
  - int números entre -2147483648 y 2147483647
  - long no acotado
  - boolean ({0,1})
- Números Reales
  - float representados en punto flotante de doble precisión
- Numeros Complejos
  - complex

Literales y expresiones

- Cada tipo tiene su forma de literales (valores constantes)
- Booleanos: True False
- Enteros: 14151234
- Reales: 14.0 3.12 0.1 .10
- Complejos: 4+3j complex(8.2, 0.99)
- La expresiones se construyen con los operadores habituales: + - * /

Expresiones Aritméticas

- Las expresiones más simples son las variables y los literales.
- Las otras expresiones se construyen usando los operadores + - * // / % **:
- Ejemplos de expresiones
  - a (variable numérica)
  - 12
  - 13.4
  - 4 + 2
  - (a + 3) / x
  - (a + 8) / (b + 2.0) * (c -3.5 -b)

Evaluando Expresiones en Python

```
>>> 4 / 3
1
>>> 4.0 / 3
1.3333333333333333
>>> 4.9 + 0.1
5.0
>>> 4.9 + 0.1 / 2
4.950000000000002
>>> 4.9 + 0.1 // 2
4.900000000000004
>>> (4.9 + 0.1) / 2
2.5
>>> (4.9 + 0.1) // 2
2.0
```

Conversión implícita (coerción)

- Si en una expresión aparece al menos un operando real, todo los otros operandos se transforman a real.
- Si se quiere evaluar 4 + 5.3 se hace 4.0 + 5.3
- Esta transformación se llama **coerción**: un valor es forzado a cambiar de tipo automáticamente.
- Python provee operadores de conversión explícitos: float(), long(), bool(), int(), complex()

Tipo de una expresión

- El tipo de una expresión puede ser consultado con el operador type()

```
>>> type(4+3)
<type 'int'>
>>> type (4.0 - 8 * 3)
<type 'float'>
>>> type(99999999999999999999999999)
<type 'long'>
```

Comparaciones

- Los operadores de comparación dan como resultado un boolean, estos son:

  < menor

  <= menor o igual

  > mayor

  >= mayor o igual

  <>, != distinto

  == igual

- Se pueden encadenar: x < y <= z se interpreta como (x < y) and (y <= z)

Ejemplos de comparaciones

```
>>> 4 < 8
True
>>> 8 <= 4.0
False
>>> 4 == 4.0
True
>>> 3 < 7 >= 2
True
>>> 1+4j < 8
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: no ordering relation is defined for complex numbers
```

Operadores booleanos

- Son operadores que permiten construir condiciones compuestas
- Todos los números se interpretan como booleanos: el 0 es False y todos los demás son True
- Los operadores and y or se evalúan en modalidad **perezoza** de izquierda a derecha

A	B	A and B	A or B	not A
True	True	True	True	False
True	False	False	True	False
False	True	False	True	True
False	False	False	False	True

Precedencia y asociatividad de operadores

- or
- and
- not
- <, <=, >, >=, <>, !=, ==
- +, -
- /, //, %
- +x, -x, ~x
- **
- Operadores de igual precedencia asocian de izquierda a derecha (excepto comparadores que **encadenan** de izquierda a derecha)

Ejemplos

- not a < b or c <> x +1
- a = b + 2 * 3 and x <= 4 or b == 0
- not 3 + 4 < 8 != 3 * 6 + 2 and 8 > x

Cadenas de caracteres

- Las cadenas de caracteres (**strings**) se escriben con comillas dobles o simples

```
>>> 'spam eggs'
'spam eggs'
>>> 'doesn\'t'
"doesn't"
>>> "doesn't"
"doesn't"
>>> '"Yes," he said.'
'"Yes," he said.'
>>> "\"Yes,\" he said."
'"Yes," he said.'
>>> '"Isn\'t," she said.'
'"Isn\'t," she said.'
```

Cadenas y líneas

- Existe un carácter especial: **fin de línea**
- Es un carácter de control que produce un cambio de línea en la salida
- Se representa como \n

```
>>> cadena = "hola\nque tal"
>>> cadena
'hola\nque tal'
>>> print cadena
hola
que tal
```

Continuación de cadena

- Una cadena se puede escribir en varias líneas sin incluir fin de línea:

```
>>> cad ="hola que tal\
... todo bien\
... saludos"
>>> cad
'hola que taltodo biensaludos'
>>> cad ="hola que tal\n\
... todo bien\n\
... saludos"
>>> cad
'hola que tal\ntodo bien\nsaludos'
```

Comillas triples

- Las cadenas pueden encerrarse entre 3 comillas simples o dobles.
- En ese caso se incluyen los fines de línea

```
>>> cad
'hola que tal\ntodo bien\nsaludos'
>>> cad = '''hola
... que tal
... todo bien'''
>>> cad
'hola\nque tal\ntodo bien'
```

Modo textual (raw)

- Si la cadena es precedida por el carácter r se interpreta como **raw string**
  - **facilita la inclusión de caracteres anidados como que normalmente tienen significados como delimitadores y comienzos de secuencias de escape.**
- Se pueden escribir varias líneas pero se requiere carácter de continuación

```
>>> cad = r"hola que tal\
... todo bien"
>>> cad
'hola que tal\\\ntodo bien'
>>>
```

Concatenación y repetición

- Las cadenas permiten las operaciones de:
- concatenación: pegar dos cadenas, con el operador +
- repetición: repetir muchas veces la misma cadena, con el operador *

```
>>> word = 'Help' + 'A'
>>> word
'HelpA'
>>> '<' + word*5 + '>'
'<HelpAHelpAHelpAHelpAHelpA>'
```

Concatenación de literales

- Dos cadenas literales consecutivas se concatenan automáticamente
- Solo funciona con literales

```
>>> 'str' 'ing'                    #  <-  This is ok
'string'
>>> 'str'.strip() + 'ing'    #  <-  This is ok
'string'
>>> 'str'.strip() 'ing'      #  <-  This is invalid
  File "<stdin>", line 1, in ?
    'str'.strip() 'ing'
```

Índices de cadenas

- Los caracteres de una cadena pueden obtenerse utilizando la operación de indizado
- Se pueden obtener rebanadas(**slices**) utilizando

```
>>> word[4]
'A'
>>> word[0:2]
'He'
>>> word[2:4]
'lp'
>>> word[:2]    # The first two characters
'He'
>>> word[2:]    # Everything except the first two characters
'lpA'
```

Inmutabilidad de las cadenas

- **No es posible cambiar parcialmente una variable cadena**

```
>>> word[0] = 'x'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
>>> word[:1] = 'Splat'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support slice assignment
```

Índices negativos

- Los índices negativos permiten contar desde el final de la cadena

```
>>> word[-1]     # The last character
'A'
>>> word[-2]     # The last-but-one character
'p'
>>> word[-2:]    # The last two characters
'pA'
>>> word[:-2]    # Everything except the last two characters
'Hel'
```

Esquema para rebanadas

- Un diagrama que ilustra como se comportan las rebanadas:
  - Los índices se asocian con **bordes** de las celdas
  - La rebanada de i a j consiste de todos los caracteres comprendidos entre los bordes i , j

Largo de una cadena

```
>>> s = 'supercalifragilisticexpialidocious'
>>> len(s)
34
>>> s[len(s)-1]
's'
>>> s[-1]
's'
```

Subcadena

El *método* `count` calcula la cantidad de veces que aparece una cadena dentro de otra:

```
>>> cadena = 'palabramacabra'
>>> cadena.count('a')
6
>>> cadena.count('ra')
2
```

El operador `in` verifica si una cadena aparece dentro de otra:

```
>>> 'brama' in cadena
True
>>> 'broma' in cadena
False
```

Estructura de selección

- Importa la ubicación (identació) de las sentencias. Ejemplo:

```
a = 10
b = 20
if a > b:
    print ("El mayor es a")
else:
    print ("El mayor es b")
```

Selección de varias alternativas

```
a, b, c = 10, 20, 30


if (a > b) and (a > c):
    print("El mayor es a")
elif (b > c) and (b > a):
    print("El mayor es b")
elif (c > a) and (c > b):
    print("El mayor es c")
else:
    print("algunos de los numeros son iguales")
```

Estructuras de iteración

- for:

```
for i in range(1, 6):
    print(i)
```

- while:

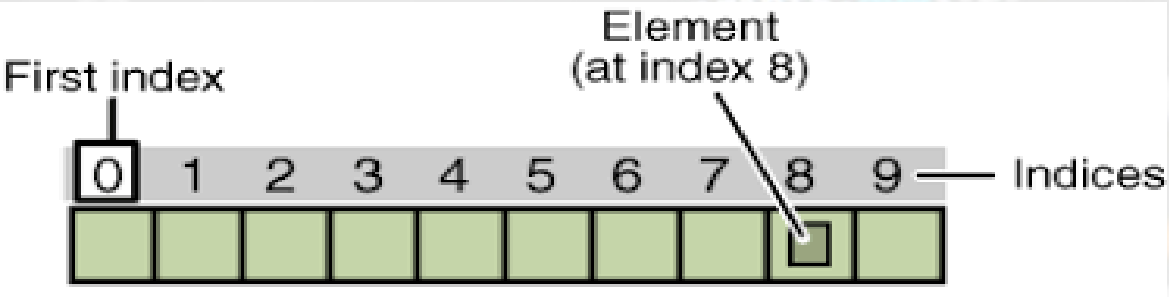
```
i = 1
while i < 6:
    print(i)
    i = i + 1
```

Funciones

```
def factorial (n):
    f = 1
    for i in range(1,n+1):
        f = f * i
    return f
```

Secuencias

- La estructura de datos más básica en Python es la secuencia.
- A cada elemento de una secuencia se le asigna un número: su posición o índice.
- El primer índice es cero, el segundo índice es uno, y así sucesivamente.



Secuencias(II)

- Python tiene seis tipos de secuencias integradas, pero las más comunes son listas y tuplas.
- Hay ciertas cosas que puedes hacer con todos los tipos de secuencia. Estas operaciones incluyen indexar, dividir, agregar, multiplicar y verificar si contiene cierto elemento.
- Además, Python tiene funciones integradas para encontrar la longitud de una secuencia y para encontrar sus elementos más grandes y más pequeños.

Listas

- La **lista** es un tipo de datos más versátil disponible en Python que se puede escribir como una lista de valores separados por comas (elementos) entre corchetes.
- Lo importante de una lista es que los elementos de una lista no necesitan ser del mismo tipo.
- Crear una lista es tan simple como poner diferentes valores separados por comas entre

Listas(II)

```
list1 = ['physics', 'chemistry', 1997, 2000]

list2 = [1, 2, 3, 4, 5 ]

list3 = ["a", "b", "c", "d"]
```

Acceso a valores en listas

- Para acceder a los valores en las listas, hay que usar los indexes entre corchetes para obtener el valor disponible en ese índice:

```
list1 = ['physics', 'chemistry', 1997, 2000];
list2 = [1, 2, 3, 4, 5, 6, 7 ];
print "elemento en el indice 0: ", list1[0]
print "elementos entre los indices 1 y 5: ", list2[1:5]
```

Acceso a valores en listas(II)

elemento en el indice 0: physics

elementos entre los indices 1 y 5: [2, 3, 4, 5]

Actualización de listas

- Se puede actualizar elementos individuales o múltiples de listas utilizando el operador de asignación:
  - list[2] = 2001
- Y puede agregar elementos a una lista con el método append ():
  - list = ['physics', 'chemistry', 1997, 2000];
  - list.append(2015)
  - print list
  - ['physics', 'chemistry', 1997, 2000, 2015]

Eliminar elementos de lista

- Para eliminar un elemento de lista, puede usar la declaración *del* si sabe exactamente qué elemento (s) está eliminando:

```
del list1[2]
```

- o el método remove() si no sabe:

```
list = [1, 2, 3]
list.remove(2)
print list
    [1, 3]
```

Operaciones básicas de lista

Python Expression	Results	Description
len([1, 2, 3])	3	Length
[1, 2, 3] + [4, 5, 6]	[1, 2, 3, 4, 5, 6]	Concatenation
['Hi!'] * 4	['Hi!', 'Hi!', 'Hi!', 'Hi!']	Repetition
3 in [1, 2, 3]	True	Membership
for x in [1, 2, 3]: print x,	1 2 3	Iteration

Funciones y métodos

Python incluye las siguientes funciones de lista:

cmp(list1, list2): compara elementos de ambas listas.

len(lista): da la longitud total de la lista.

max(lista): devuelve el artículo de la lista con el valor máximo.

min(lista): devuelve el artículo de la lista con valor mínimo.

list(seq) : convierte una tupla en una lista.

Funciones y métodos(II)

list.append (obj): anexa objeto obj a la lista

list.count (obj): devuelve el recuento de cuántas veces se produce obj en la lista

list.extend (seq): añade el contenido de seq a la lista

list.index (obj): devuelve el índice más bajo en la lista en que aparece el obj

list.insert (index, obj): inserta objeto obj en lista en índice de desplazamiento

list.pop (obj = list [-1]): elimina y devuelve el último objeto u obj de la lista

list.remove (obj): elimina objetos obj de la lista

list.reverse (): invierte los objetos de la lista en su lugar

list.sort ([func]): ordena los objetos de la lista, use func de comparación si se le da

Tuplas

- Una tupla es una secuencia de objetos de Python inmutables.
- Las tuplas son secuencias, como listas.
- Las diferencias entre tuplas y listas son que las **tuplas no se pueden cambiar** a diferencia de las listas y las tuplas usan paréntesis, mientras que las listas usan corchetes.
- Crear una tupla es tan simple como poner diferentes valores separados por comas.
- Opcionalmente puede poner estos valores separados por comas entre paréntesis también

Tuplas(II)

```
tup1 = ('physics', 'chemistry', 1997, 2000)

tup2 = (1, 2, 3, 4, 5 )

tup3 = "a", "b", "c", "d"
```

Tuplas(III)

- La tupla vacía está escrita como dos paréntesis que no contienen nada

```
tup1 = ()
```

- Para escribir una tupla que contenga un solo valor, debe incluir una coma, aunque solo haya un valor:

```
tup1 = (50,)
```

Diccionarios

- {} denota el diccionario vacío, no el conjunto vacío.
- Un diccionario es como una lista pero indexada por claves elegidas por el usuario, que son miembros de cualquier tipo inmutable. En realidad es un conjunto de pares "clave:valor".

Diccionarios - Ejemplos

- Diccionario de números junto con sus cuadrados

  {2: 4, 4: 16, 6: 36}

- Días hábiles de la semana:

  {"lunes":2, "martes":3, "miercoles":4;"jueves":4; "viernes":5; }

Diccionarios – Obtener valores

```
dict_0 = {'color': 'green', 'points': 5}
```

- Valor asociado con su clave

```
print(dict_0['color'])
print(dict_0['points'])
```

- Valor asociado con get()

```
color = dict_0.get('color')
points = dict_0.get('points', 0)
print(color )
print(points)
```

Diccionarios – Agregar y borrando valores

```
dict_0 = {'color': 'green', 'points': 5}
    dict_0['x'] = 0
    dict_0['y'] = 25

    del dict_0['point']
```

Diccionarios – Listar valores

```
fav_languages = {
    'juan': 'python',
    'sara': 'c',
    'eduardo': 'ruby',
    'felipe': 'python',
    }
# Show each person's favorite language.
for name, language in fav_languages.items():
    print(name + ": " + language)
```

Manejo de archivos

- En Python, para abrir un archivo usaremos la función **open**, que recibe el nombre del archivo a abrir.

```
archivo = open("archivo.txt")
```

- La operación más sencilla a realizar sobre un archivo es leer su contenido.
- Para procesarlo línea por línea, es posible hacerlo de la siguiente forma:

```
linea=archivo.readline()
while linea != '':
    linea=archivo.readline() # procesar línea
```

Manejo de archivos(II)

- La siguiente estructura es una forma equivalente a la vista en el ejemplo anterior.

```
for linea in archivo:
    # procesar línea
```

Manejo de archivos(III)

- Es posible, además, obtener todas las líneas del archivo utilizando una sola llamada a función:

```
lineas = archivo.readlines()
```

Manejo de archivos(IV)

- Para cerrar un archivo simplemente se debe llamar a:

```
archivo.close()
```

Ejemplo

Este ejemplo cuenta la cantidad de caracteres que tiene cada linea del archivo.

```
archivo = open("archivos.txt")
i = 1
for linea in archivo:
    print "la linea ", i," tiene ",  len(linea)," caracteres"
    i=i+1
archivo.close()
```

Ejemplo – Escribiendo y agregando en un archivo

Writing to a file

```
filename = 'journal.txt'
with open(filename, 'w') as file_object:
file_object.write("I love programming.")
```

Appending to a file

```
filename = 'journal.txt'
with open(filename, 'a') as file_object:
file_object.write("\nI love making games.")
```