Bioinformática avanzada: problemas y algoritmos

Rodrigo Santamaría

2014

Bioinformática para adultos

Rodrigo Santamaría

2014

Por qué ser adultos?

INTRODUCCIÓN

Enfoque

- Cualquiera sabe hacer un BLAST
  - Las soluciones disponibles no son siempre
    - Suficientes
    - Adaptadas
    - Eficientes
    - Rentables

El problema inmobiliario

Comprar una casa

- Caro
- Poco configurable
- Poco aprendizaje
- Rápido
- Bien establecido

Construir una casa

- Barato
- Muy configurable
- Mucho aprendizaje
- Lento
- Novedoso

WALDEN

HENRY DAVID THOREAU

errata naturae

El problema

Usar

- Comprar
- Sistema operativo cerrado
- Programa de terceros

dependencia
precio
flexibilidad
dificultad
rendimiento
estabilidad

Crear

- Construir
- Sistema operativo abierto
- Programa nuestro

dependencia
precio
flexibilidad
dificultad
rendimiento
estabilidad

Nobody expected the Spanish Inquisition

PYTHON

Python

- Es un lenguaje de programación
  - Forma de comunicación (adulta) con el ordenador
- 'Fácil' de entender para no expertos
- Cada vez más extendido en bioinformática
  - Biopython
  - Anaconda

Python se llama así por los Monty Python, grupo cómico inglés imprescindible: https://www.youtube.com/watch?v=na3i3O8aWv8

Python

- Sintaxis limpia
- Tipado dinámico
- Interpretado
- Abierto
- Multiparadigma

Recursos

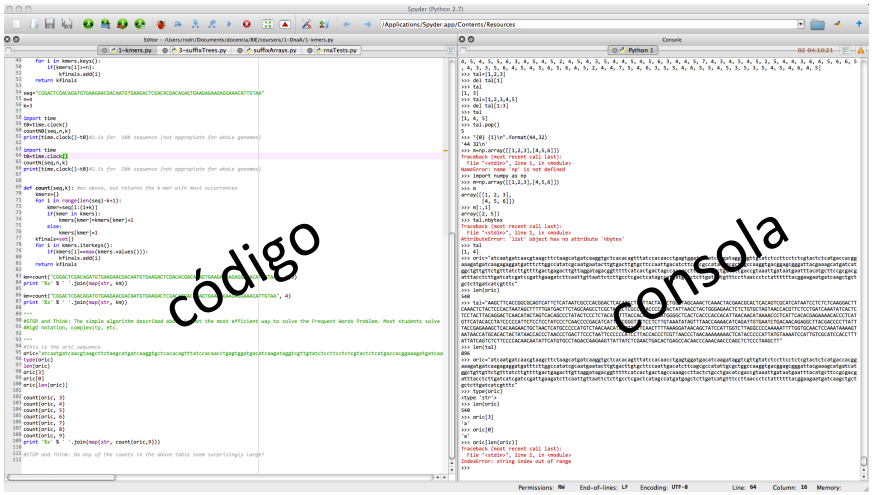
- **Tutorial** de Python en CodeAcademy
  - http://www.codecademy.com/es/tracks/python-latinamerica
- **Chuleta** de Python
  - http://vis.usal.es/rodrigo/documentos/bioinfo/avanzada/pythonCheatsheet.pdf
- **Internet**
  - Internet es tu amigo: busca!
- **Tutorial** de Algoritmos Bioinformáticosen Coursera
  - https://www.coursera.org/course/bioinformatics

Python - Herramientas

- Python: http://www.python.org/downloads/
- Entornos de desarrollo (IDE):
  - Spyder:
    - https://code.google.com/p/spyderlib/
    - Incluye intérprete, lo usaremos en clase
  - PyCharm:
    - http://www.jetbrains.com/pycharm/download/

Comenzando

- Los programas en Python se guardan en archivos ***.py***
- Spyder
  - Podemos ejecutar un programa entero o sólo un bloque de líneas 
  - Dos ventanas:



De las que os harán libres

VARIABLES

Cadenas (str)

```
oric="atcaatgatcaacgtaagcttctaagcatgatcaaggtgctcacacagtttatccacaacctgagtggatg
acatcaagataggtcgttgtatctccttcctctcgtactctcatgaccacggaaagatgatcaagagaggatgattt
cttggccatatcgcaatgaatacttgtgacttgtgcttccaattgacatcttcagcgccatattgcgctggccaagg
tgacggagcgggattacgaaagcatgatcatggctgttgttctgtttatcttgttttgactgagacttgttaggata
gacggtttttcatcactgactagccaaagccttactctgcctgacatcgaccgtaaattgataatgaatttacatgc
ttccgcgacgatttacctcttgatcatcgatccgattgaagatcttcaattgttaattctcttgcctcgactcatag
ccatgatgagctcttgatcatgtttccttaaccctctatttttacggaagaatgatcaagctgctgctcttgatca
tcgtttc"
```

```
oric          #valor
type(oric)    #tipo
len(oric)     #longitud
oric[3]       #acceso
oric[0]       #ojo con el comienzo y fin
oric[len(oric)]

oric[3:7]     #acceso a intervalo
oric[4]="X"   #modificación de lugar
```

`oric` contiene la secuencia de nucleótidos que indica un **origen de replicación** en *Vibrio cholerae*, la bacteria patógena que causa el cólera

Curiosamente, iremos viendo cómo los orígenes de replicación se pueden descubrir mediante un ordenador, sin necesidad del laboratorio!

Cadenas (str)

- `oric` es una **variable**, de tipo cadena (str)
  - `oric` es su **nombre**
    - Cuidado con nombres 'raros': no se permiten tildes, eñes, guiones, espacios, o números al comienzo
    - Sí se puede usar el guión bajo (_) o mayúsculas
  - `"atcaa…"` es su **valor**
    - Es por tener este valor que sabemos que es una cadena
  - **Ejercicio:** introducir una mutación puntual (SNP) en la posición 133 de oriC

Números

```
n=3       #un número entero
type(n)   #tipo 'int'
q=8.56    #un número real
type(q)   #tipo 'float'

n+q       #operaciones aritméticas
n-q
n*q
n/q
n**q      #n elevado a q
n%2       #operación módulo
(n+q*q)/2.0
```

- ¿Por qué debería un biólogo preguntarse cómo encontrar soluciones informáticas a problemas biológicos?
- Lo cierto es que en la biología moderna, los métodos computacionales son el único modo realista de abordar ciertas cuestiones. Por ejemplo, las aproximaciones experimentales para encontrar *oriC* en una determinada especie consumen mucho tiempo (implicaría ir probando a 'quitar' trozos del genoma al bicho en cuestión hasta que deje de replicarse). Esto hace que, a nivel experimental, *oriC* sólo esté descrito para unas pocas especies.
- Si diseñamos un buen método computacional para encontrar *oriC*, los biólogos pueden dedicar su tiempo a otras tareas más interesantes que cortar bichos!

Comentarios

- Simples anotaciones en español para entender el código
  - El intérprete las ignora

```
#comentario de una línea


"""Comentario de varias líneas:
¡Soy un leñador y me
siento con energía,
duermo toda la noche,
y trabajo todo el día!"""
```

Ejercicio

- Toma el ADN mitocondrial de humano:
  - http://vis.usal.es/rodrigo/documentos/bioinfo/filogenia/mitDNAprimates.fasta
- Almacénalo en una variable `seq`
- ¿Cuáles son los valores de los 10 nucleótidos en el centro de la secuencia?

Sol.: `'CCGGGTTTTC'`

Ejercicio

- Ahora cambia el valor de `seq` para que contenga el ADN mitocondrial del gorila
- ¿Cuál es ahora la cadena central de 10 nucleótidos?

Sol.: `'CCGGGTTTAC'`

Una vez que tenemos la solución a un problema, es fácil aplicarlo a otras secuencias, organismos, etc.!

Esta es una de las grandes ventajas de la informática respecto a la experimentación en laboratorio, la facilidad de replicar o extender métodos.

Salida en consola

```
print oric
print “el origen de replicación es:\n”+oric

print "el comienzo de oric es:\n"+oric[0:10]+"\ny el
final:\n"+oric[len(oric)-10:len(oric)-1]


print n +“ y “ +q   #error! deben imprimirse cadenas

print str(n) +“ y “ +str(q)   #conversión de tipos

print “{0} y {1}“.format(n, q) #formato
```

Llevando el programa por donde queremos

CONTROL DE FLUJO

Condiciones

- **Control de flujo**: nos permite obtener distintos resultados según lo que esté ocurriendo en el programa
- **Comparadores:**

01. Igual a (`==`) *

02. No es igual a (`!=`)

03. Menor que (`<`)

04. Menor o igual que (`<=`)

05. Mayor que (`>`)

06. Mayor o igual que (`>=`)

* Fíjate que == se usa para comparar si dos objetos son iguales y que = se usa para asignar valor a una variable

Comparaciones

- El resultado de una comparación puede ser `True` o `False`
  - La comparación será cierta o no (lógica *booleana*)

```
7==8
8>3
“a”!=“b”
“manuel”==“sonia”
n<=3

print “{0} y {1} son iguales? --> 2“.format(n, q, n==q)
```

Ejercicio

- Predice el resultado de estas comparaciones:
  ① `20 + -10 * 2 > 10 % 3 % 2`
  ② `(10 + 17)**2 == 3**6`
  ③ `1**2**3 <= -(-(-1))`
  ④ `40 / 20 * 4 >= -4**2`
  ⑤ `100**0.5 != 6 + 4`
- Imprime por pantalla si el primer y el último nucleótido de *oriC* son iguales
  - Y el segundo y el penúltimo?

Operadores booleanos

- Son palabras utilizadas para unir sentencias de Python gramaticalmente correctas
  - `and`, "y" en español
  - `or`, "o" en español
    - Una cosa O la otra o AMBAS
  - `not`, "no" en español

```
1>2 and 2<3
1<2 and 2<3

1<2 or 2>3
1>2 or 2>3

not False
not 40>41
```

Operadores booleanos

```
"""
     Operadores booleanos
----------------------------
True and True es True
True and False es False
False and True es False
False and False es False

True or True es True
True or False es True
False or True es True
False or False es False

Not True es False
Not False es True

"""
```

`and` da como resultado `True` sólo si las **expresiones** a *ambos* lados de `and` son verdaderas (`True`)

`or` da como resultado `True` cuando *ambas* (es decir, una, la otra, *¡o las dos!*) **expresiones** a cada lado de `or` son verdaderas (`True`)

`not` da como resultado `True` para sentencias False y `False` para sentencias True

Ejercicio

- Predice si estas expresiones retornarán `True` o `False`

① `-(-(-(-2))) == -2 and 4 >= 16**0.5`
② `19 % 4 != 300 / 10 / 10 and False`
③ `-(1**2) < 2**0 and 10 % 10 <= 20 - 10 * 2`
④ `True and True`

① `2**3 == 108 % 100 or 'Quijote' == 'King Arthur'`
② `True or False`
③ `100**0.5 >= 50 or False`
④ `True or True`
⑤ `1**100 == 100**1 or 3 * 2 * 1 != 3 + 2 + 1`

① `not True`
② `not 3**4 < 4**3`
③ `not 10 % 3 <= 10 % 2`
④ `not 3**2 + 4**2 != 5**2`
⑤ `not not False`

① False
② True
③ True
④ True
⑤ False

① True
② True
③ False
④ True
⑤ False

① False
② False
③ True
④ True

Operadores booleanos

- Podemos combinar varias operaciones mediante el uso de *paréntesis*
  - Si no se usan, la prioridad es:

01. primero se calcula `not`;

02. después se calcula `and`;

03. por último se calcula `or`.

- Predice los siguientes resultados:

① `False or not True and True`
② `False and not True or True`
③ `True and not (False or False)`
④ `not not True or False and not True`
⑤ `False or not (True and True)`

`if`, else y elif

- `if` ejecuta un **bloque de código** si la **expresión** que evalúa es `True`
- Sintaxis:

```
if 8 < 9:
  print "¡Ocho es menor que Nueve!"
```

  - La **expresión** `8 < 9` va separada del `if` por un espacio y termina con **:**
  - El **bloque de código** serán una o más líneas indentadas respecto al `if`

if, else y elif

- `else` ejecuta un **bloque de código** si la **expresión** que evaluó if es `False`
- Sintaxis:

```
if 8 < 9:
  print "¡Ocho es menor que Nueve!"
else:
  print "OMG, ¡Ocho NO es menor que Nueve!"
```

  - No lleva asociada ninguna expresión, pero también debe terminar con **:**
  - El **bloque de código** de `else` también serán una o más líneas indentadas

if, else y elif

- `elif` es como `else`, pero añade una nueva expresión que debe ocurrir para ejecutar su **bloque de código**
- Sintaxis:

```
if 8 < 9:
  print "¡Ocho es menor que Nueve!"
elif flipando==False:
  print "OMG, no estoy flipando, ¡Ocho NO es menor
         que Nueve!"
```

- Se pueden **anidar** bucles if/else/elif como se quiera, pero recuerda, ¡la *indentación* es clave!

Ejercicio

- Si el primer nucleótido de *oriC* es igual a `'a'`, muestra por pantalla su segundo nucleótido
  - Si no lo es, si la longitud de *oriC* es menor que 600 nucleótidos y es múltiplo de 9, imprime los 50 últimos nucleótidos de *oriC*
  - Finalmente, si tampoco se da la condición anterior, copia en la variable `nuc` la posición 37 de oric
  - Prueba que todas las opciones funcionan cambiando la primera condición a una `'g'` y/o comprobando si es múltiplo de 8

Organizando el código

FUNCIONES

Anuncio de servicio público

- En clase aprenderemos mucho de Python, pero pensar que ya lo tenemos todo es como pensar que después de estudiar alemán durante un año no necesitas un diccionario si viajas allí
- En Python tenemos muchos recursos que nos ayudarán a recordar la sintaxis, palabras clave o refrescar conceptos
  - Documentación oficial: http://docs.python.org/3/
  - Pequeña chuleta de fabricación propia:
    - http://vis.usal.es/rodrigo/documentos/bioinfo/avanzada/pythonCheatsheet.pdf

Funciones

- Una función es una sección de código **reutilizable**, que realiza una **tarea específica**

- ¿Por qué usar funciones en vez de un bloque gigante de código?
  1. Si algo sale mal, es más fácil encontrar y arreglar errores si el programa está bien organizado
  2. Evita reescribir bloques de código muy utilizados

Funciones

- Sintaxis

```
#Conviene añadir una descripción
def nombreFuncion(argumentos):
    linea1
    …
    linea n
    return valor
```

```
#añade el IVA 'normal'
def aplicarIVA(factura):
    factura *=1.21
    print factura
    return factura
```

- Como en el bloque `if`, usamos `:` e indentación

Funciones

- Llamada a funciones

```
factura=aplicarIVA(35.50)
print factura

factura=aplicarIVA()   #dará un error ya que 'espera' un argumento


#ya hemos usado alguna función sin saberlo!
length=len("Ph nglui Mglw nafh Cthulhu R lyeh wgah nagl fhtagn.")
print length
```

- Podemos definir y usar varios argumentos, separados por comas, o no poner ninguno

Funciones

- Funciones que llaman a funciones

```
def amor_con_amor(n):
    return n + 1

def se_paga(n):
    return n + 2
```

```
#equivalente a la función anterior!
def se_paga(n):
    n=amor_con_amor(n);
    return amor_con_amor(n)
```

Ejercicio

- Escribid una función `mitad` que retorne los 10 nucleótidos en el centro de una cadena que acepte como argumento:
  - **entrada**: cadena
  - **salida**: cadena del centro
- Probad la función con *oriC*
- Modificad la función anterior para que le podamos indicar longitud de la cadena central
  - **entrada**: cadena, n
  - **salida**: cadena del centro

variables a tutiplén

LISTAS Y DICCIONARIOS

Listas

- Es un **tipo de variable** que almacena una **colección** de valores
- Asignación

```
listaVacia=[]

wild_animals=[“perezoso”, “ornitorrinco”, “lemur”]

bonoLoto=[3,0,8,1,4,99]
```

Listas

- Acceso:

```
wild_animals[2]

bonoLoto[3] + bonoLoto[0]

print "animal {0}, número {1}".format(wild_animals[1],
                                      bonoLoto[1])

wild_animals[2]="tigre"
```

- Eliminación:

```
del bonoLoto[4] #elimina el elemento en la posición 4
```

Listas

- Adición (**`append`**) y longitud (**`len`**)

```
len(wild_animals)
wild_animals.append("elefante")
len(wild_animals)
```

- Particionado (**`[a:b]`**)
  - comienza en `a` y termina *antes* de `b`

```
wild_animals[1:2]
bonoLoto[2:5]
wild_animals[:3] #si no ponemos nada, hasta inicio o fin
oric[500:]       #funciona con cadenas!
```

Listas

- Índice (**`index`**) e inserción (**`insert`**)

```
wild_animals.index("tigre")
wild_animals.insert(3, "gorila")
wild_animals

oric.index("atcg")   #también funciona con cadenas!
```

- Recorrido (**`for`**)

```
#recorremos la lista elevando al cuadrado cada elemento x
for x in bonoLoto:   #aprenderemos más sobre for pronto!
    print x*x

for base in oric:    #también funciona con cadenas!
    print base
```

Ejercicio

- Contar el número de adeninas en *oriC*
- Determinar el contenido en GC de *oriC**

El contenido en GC (guanina-citosina) es el porcentaje de bases G o C respecto al total en una secuencia de ADN

Los enlaces GC son más fuertes que los AT (tres enlaces de hidrógeno en vez de dos) y por tanto más resistentes, por ejemplo, a la desnaturalización por temperatura.

El contenido en GC varía entre organismos, desde un 20% en el *Plasmodium falciparum* hasta el 70% en algunas bacterias. De hecho, en estas últimas, a veces se utiliza para clasificarlas (bacterias con alto vs bajo GC)

*Ojo, cuidado con las divisiones: si los dos números son enteros, por ej: `5/6`, el resultado será la parte entera de la división, es decir 0 en vez de 0.83. Podemos convertir un número entero a real así: `(float)5/6`

Conjuntos

- Un conjunto es como una lista, pero no permite repeticiones de elementos

```
conjunto=set()              #un conjunto vacío
conjunto.add("bola8")       #añadir un elemento
conjunto.add("bola8")       #añadir un elemento
conjunto

lista=[1,2,4,4]
conjunto.update(lista)      #añadir desde lista
```

Diccionarios

- Son como listas, pero a los valores se accede por una **clave** en vez de por posición
- Asignación y acceso

```
dicVacio={}

#animales en riesgo de extinción: el nombre del animal
#es la clave y el número de individuos el valor
wild_animals={"amur leopard": 37, "black rhino" : 4848,
              "cross-river gorilla" : 250}
wild_animals["black rhino"]
```

https://worldwildlife.org/species/directory?direction=desc&sort=extinction_status

Diccionarios

- Inserción, modificación y borrado

```
wild_animals[“sumatran elephant”]=2600
wild_animals[“amur leopard”]=23
del wild_animals[“amur leopard”] #borrado por clave
```

Listas y Diccionarios

- Listas, diccionarios y tipos básicos se pueden **combinar** de formas tan complicadas como queramos

```
#Llévate todo esto cuando salgas de viaje!
inventario={"oro" : 500,
"zurrón" : ["piedra", "cuerda", "manzana"],
"morral" : ["daga", "flauta", "manta" , "queso"]}

inventario["bolsillo"]=["gema", "pelusa"]
```

una y otra vez

BUCLES

while

- El **bucle** `while` es similar a la sentencia `if`: ejecuta un **código** si su **condición** es `True`
- Diferencia: continúa ejecutándose **una y otra vez** mientras la condición sea `True`

```
recuento = 0
while recuento < 5:
  print "Hola, soy un bucle while y el recuento es ", recuento
  recuento+=1
```

while

```
recuento = 0
                condición
while recuento < 5 :
    print "Hola, soy un bucle while y el recuento es ", recuento
    recuento+=1
```

bloque

Dentro del **bloque**, podemos hacer lo que queramos, como en cualquier otro lugar

Es conveniente hacer algo que, eventualmente, **modifique la condición a `False`** para permitirnos salir del bucle (por ejemplo, incrementar `recuento`)

while

- Los **bucles infinitos** son aquellos cuya condición nunca se hace `False`
  - Como resultado, el programa parece que se queda "colgado"
  - Si nos ocurre, Ctrl+C deshace el entuerto
  - Hay que asegurarse que hay al menos una orden dentro del código que haga la condición `False` y que se ejecute al menos una vez

```
#un bucle infinito!!
recuento = 0
while recuento < 5:
  print "Hola, soy un bucle while y el recuento es ", recuento
```

Ejercicio

- Crea un bucle `while` que imprima los cuadrados de los números del 1 al 10
- Crear mediante un bucle `while` la cadena complementaria a *oriC*

La cadena de ADN es doble, de manera que dos secuencias de nucleótidos se entrelazan en base a enlaces químicos.
Los pares de bases A y T por un lado, y G y C por otro, son los que forman los enlaces, de modo que tenemos una secuencia de ADN como:

```
…ACGTTCGA…
|||||||||
…TGCAAGCT…
```

Es evidente que dada una de las dos secuencias enlazadas, podemos inferir la **cadena complementaria**. Esta estructura en doble hélice es, en última instancia, la responsable de que las células de tu cuerpo se multipliquen!

Ejercicio

- Organizar el código anterior de manera que tengamos una función `complementaria`:
  - **entrada**: `seq` (cadena original)
  - **salida**: cadena complementaria
- Hacer otra función `inversa`:
  - **entrada**: `seq` (cadena original)
  - **salida**: cadena inversa

Ejercicio

- Hacer una tercera función `revcomp`:
  - **entrada**: `seq` (cadena original)
  - **salida**: cadena inversa complementaria
  - *pista*: podemos invocar las funciones anteriores

Ejercicio

- Hacer una función `frecuencia`:
  - **entrada**: `seq` (cadena original)
  - **salida**: diccionario que tenga como claves los nucleótidos y como valores el número de veces que aparece en `seq`

for

- Otra forma de bucles que se lee como:

```
for i in range(10):
  print i
```

"Para cada número `i` de la serie de 0 a 9, imprime `i`"

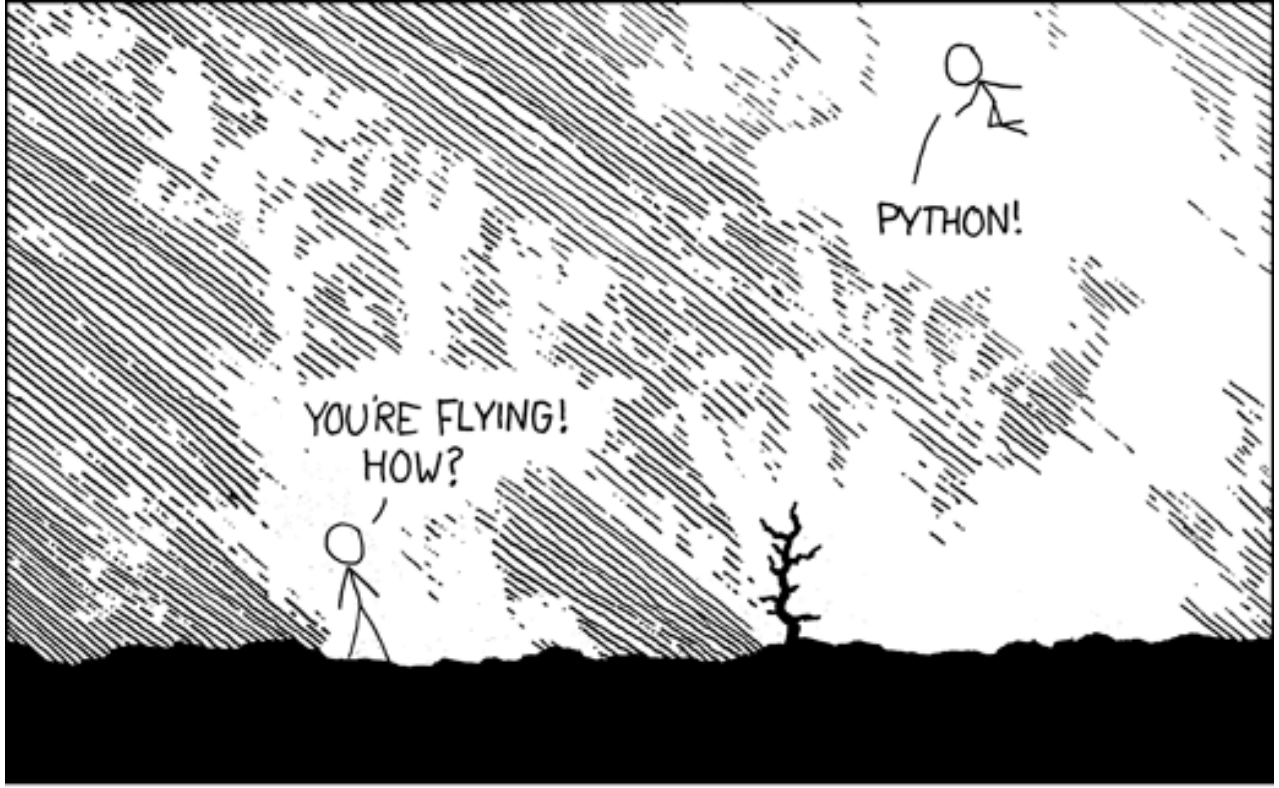
- Recuerda que vimos que `for` se puede usar también sobre cadenas, listas y diccionarios!

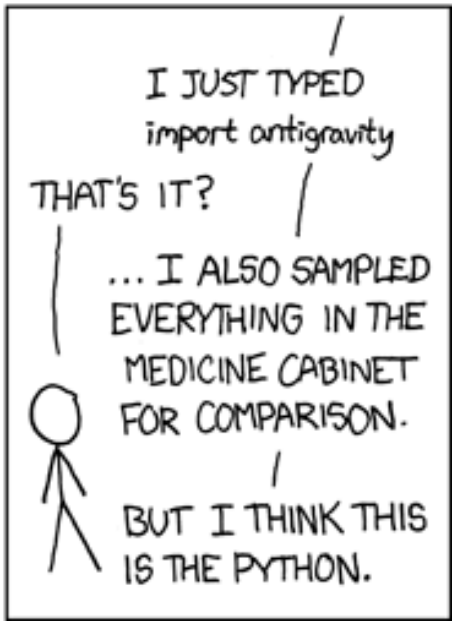
```
for i in coleccion:
  print i
```

"Para cada letra/elemento/clave `i` de la cadena/lista/diccionario `colección`, imprime `i`"

Ejercicio

- Cambia los bucles `while` en las funciones `complementaria` e `inversa` por bucles `for`
  - ¿Cuál te resulta más sencillo?





http://xkcd.com/353/