Introducción al pensamiento computacional

Objetivos:

- Aprender a resolver problemas de manera computacional
- Entender los puntos en común entre todos los lenguajes de programación
- Desarrollar las bases para una carrera en computer science

Platzi

Introducción al cómputo

La primera computadora: Creada por los griegos, su único propósito era calcular las posiciones del sol, la luna y algunas constelaciones.

El telar de Jacquar: Lo importante del dispositivo son los punch cards que eran tarjetas que representan la información que tiene se tiene que hacer la máquina para hacer un pedazo de tela bien detallado.

El motor analítico de Babbage: En el siglo XIX hecho para el cálculo matemático en donde se utilizaban engranes, que era la tecnologia mas avanzada de ese entonces, esto nos llevó a darnos cuenta que no solo podemos utilizar partes mecánicas sino que podemos separar las instrucciones del cálculo y así hacer distintos tipos de cálculos.

Máquinas para contar censos en estados unidos: Estas máquinas ayudaban a contar los censos con muchos datos en donde ya se utilizaban las punch cards.

ENIAC (Electronic Numerical Integrator and Computer): La primera computadora electrónica digital creada por Alan Turing y Alonso. Este dispositivo ya se comportaba con las reglas matemáticas, utilizaba el sistema decimal y para su funcionamiento se tenían que conectar y desconectar los cables de manera distinta.

Von Neumann, EDVAC(Electronic Discrete Variable Automatic Computer): Von Neumann crea su arquitectura para realizar el cómputo y almacenar el programa que va a ejecutar la computadora junto con los datos. Es aquí donde surge la primera computadora en donde se podía almacenar en memoria el programa a correr EDVAC.

Microchips: Nos dieron la pauta para llegar al cómputo en el cual nos encontramos dentro de las primeras computadoras hechas con chip encontramos a la apple1. Iniciamos a hacer electrónica hecha con luz para generar estructuras microscópicas.

Cómputo cuántico de Feymann: Aporta las bases del cómputo cuántico

Platzi

Introducción a los lenguajes de programación

¿Cómo damos instrucciones?

- Conocimiento declarativo e imperativo
- Algoritmos

Conocimiento declarativo: Solamente nos dice el tipo de relaciones que se tiene entre variables u objetos. Por ejemplo una fórmula matemática

Conocimiento imperativo: Nos dice como llegar a un resultado. Dentro de esta rama encontramos a los algoritmos

Algoritmo: Un algoritmo es una lista finita de instrucciones que describen un cómputo, que cuando se ejecuta con ciertas entradas (inputs) ejecuta pasos intermedios para llegar a un resultado (output)

Los algoritmos son la base de los primeros lenguajes de programación. (C, C++, php, python)

La primera persona en hacer un programa de cómputo fue Augusta Ada quien hace el algoritmo para resolver los números de Bernoulli.
Las computadoras entienden solo números binarios por lo que para dar una instrucción a una computadora debemos escribir una secuencia de ceros y unos. Programar de esta manera es muy difícil por lo que se crearon los primeros lenguajes de programación en lo que se hacía más fácil la comunicación de la computadora y los humanos.
Para llevar a cabo este proceso en el que hacíamos más fácil dar instrucciones a la computadora, surgieron personajes como **Grace Hopper** quien crea el sistema de traducir el lenguaje parecido al del humano hacia el lenguaje que entienden las computadoras.

Siguiendo la misma idea de Hopper surge el lenguaje moderno llamado c con el profesor Ritchie, este lenguaje es el padre de muchos otros lenguajes.

En los 90 Guido van Rossum crea Python, un lenguaje mucho más fácil de entender y programar para el ser humano.

Programación:

- Turing complete: para implementar cualquier algoritmo
- Los lenguajes de programación modernos dan primitivos que son más convenientes que los primitivos de Turing.

Lenguajes:

- **Sintaxis:**
  Define la secuencia de los símbolos que está bien formada
- **Semántica estática:**

Platzi

Define qué enunciados con sintaxis correcta tienen significado

- **Semántica:**
  Define el significado. En los lenguajes de programación solo hay un significado.

Platzi

Elementos básicos de Python

Lenguajes de programación:

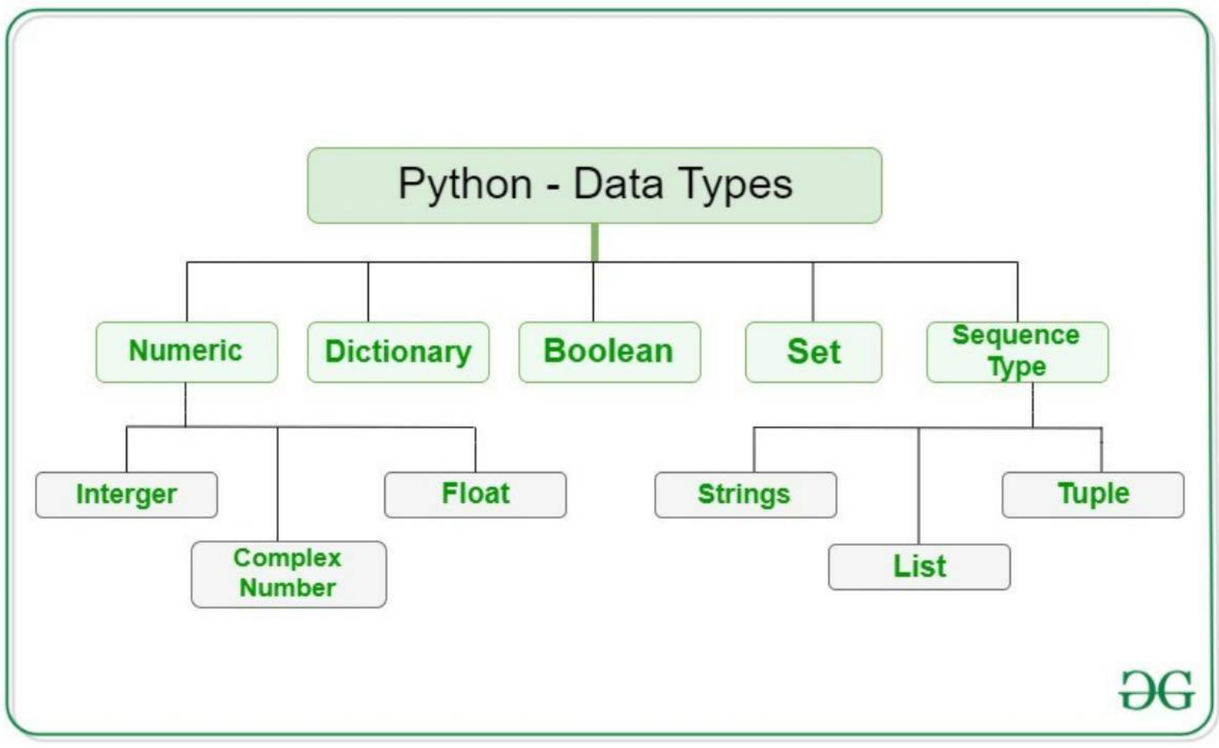
- **De bajo nivel:** Significa que está optimizado para que una máquina pueda entenderlo (se parece más a los unos y ceros)
- **De alto nivel:** son aquellos lenguajes que se parecen más a la forma en la que nos comunicamos los humanos.
- **Lenguaje general:** Significa que tiene todos los primitivos que nos otorga Turing para poder implementar y computar cualquier tipo de algoritmo.
- **Lenguaje de dominio específico:** Son lenguajes especializados que están tarjeteados a aplicaciones muy específicas
- **Lenguajes interpretados:** Después de que se ejecuta el programa las instrucciones se traducen al lenguaje máquina
- **Lenguajes compilados:** Convertimos antes de entregárselo a la computadora el lenguaje en un lenguaje máquina

Python es un lenguaje de alto nivel, general y es interpretado.

Elementos básicos:
Literales: Las maneras de iniciar objetos directamente en memoria de manera que las anotamos directamente en código ejemplo: 1, abc, 2.0, True
Operadores: Con ellos podemos sumar, multiplicar, utilizar módulos, igualdades, potencias etc

Objetos: Son la abstracción más alta dentro de los lenguajes de programación, la forma con la que modelamos el mundo dentro de nuestros programas.

Tipos de objetos: int, flotantes, cadenas, Booleanos.



Platzi

Asignación de variables

Variables: En programación, una variable está formada por un espacio en el sistema de almacenaje (memoria principal de un ordenador) y un nombre simbólico (un identificador) que está asociado a dicho espacio. Regularmente vinculamos a las variables con los valores con el símbolo de igual “=”

Es buena práctica dar nombres a las variables relacionados a lo que hace nuestro programa

Variables en Python

- Pueden contener mayúsculas, minúsculas, números (sin comenzar con uno) y el símbolo _
- No pueden llamarse igual que las palabras reservadas

Palabras reservadas

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

Platzi

Cadenas y entradas

Cadenas: En programación, una **cadena de caracteres** o **frases** (*string*, en inglés) es una secuencia ordenada (de longitud arbitraria, aunque finita) de elementos que pertenecen a un cierto lenguaje formal o alfabeto análogas a una fórmula o a una oración. En general, una cadena de caracteres es una sucesión de caracteres (letras, números u otros signos o símbolos). Si no se ponen restricciones al alfabeto, una cadena podrá estar formada por cualquier combinación finita de los caracteres disponibles (las letras de la 'a' a la 'z' y de la 'A' a la 'Z', los números del '0' al '9', el espacio en blanco ' ', símbolos diversos '!', '@', '%', etcétera).

Las cadenas deben ir entre comillas simples o dobles, si escribes algo dentro de comillas puede considerarse una cadena.

```
>>> '123'
'123'
```

Cuando escribes una cadena y a esta le pre sigue el signo asterisco y después un número, la cadena se repetirá el número de veces del valor del número escrito después del asterisco.

```
>>> '123' * 3
'123123123'
```

Cuando utilizamos el operador de suma entre cadenas, lo que haremos es concatenar éstas dos cadenas.

```
>>> '123' + '456'
'123456'
```

Nosotros podemos mezclas éstos operadores para realizar distintos tipos de computo por ejemplo:

```
>>> ('hip ' * 3) + ' ' + 'hurra!'
'hip hip hip  hurra!'
```

Otra forma más legible es hacerlo con formato con el uso de la f:

```
>>> f'{"Hip " * 3} hurra'
'Hip Hip Hip  hurra'
```

Platzi

Cadenas (strings)

- len(longitud): la función len nos devuelve el tamaño de nuestra cadena.
  por ejemplo:
  La siguiente cadena ‘palabra’ contiene 7 letras, y lo podemos saber con la funcion len()

  ```
  >>> cadena = 'palabra'
  >>> len(cadena)
  7
  ```

- Indexing (indexación): Para ubicar cada elemento de la cadena mediante sus índices.

  Para ubicar la posición de cada caracter o letra dentro de una cadena podemos usar la indexación mediante el uso de corchetes.
  ejemplo:
  ```
  >>> cadena[0]
  'p'
  ```

- slicing(rebanadas): cortar las cadenas con los elementos que necesitemos
  ejemplo: queremos ‘abra’ de la cadena palabra lo hacemos de la siguiente manera, recordemos que las posiciones inician desde el cero y no desde el uno.

  ```
  >>> cadena[3:]
  'abra'
  ```

- Para concatenar cadenas solo utilizamos el operador +
  ejemplo:
  ```
  >>> 'Quiero decir una ' + cadena
  'Quiero decir una palabra'
  ```

  esto pasa porque en la variable cadena tenemos guardado el string palabra.

- Si queremos repetir una cadena podemos hacer lo siguiente

  ```
  >>> f' hola, ' * 10
  ' hola,  hola,  hola,  hola,  hola,  hola,  hola,  hola,  hola,  hola, '
  ```

  o lo siguiente:

  ```
  >>> 'hola, ' * 10
  'hola, hola, hola, hola, hola, hola, hola, hola, hola, hola, '
  ```

Platzi

Entradas

- Python tiene la función input para recibir datos del usuario del programa.
- Input siempre regresa cadenas, por lo que si queremos utilizar otro tipo, tenemos que hacer type casting.

Ejemplo de entrada con input:

```
>>> nombre = input("Cual es tu nombre: ")
Cual es tu nombre: Aldo
>>> print(nombre)
Aldo
```

O también podemos imprimir con formato:

```
>>> print(f'Tu nombre es {nombre}')
Tu nombre es Aldo
```

Si queremos recibir un número del usuario debemos de hacer typecasting esto lo logramos colocando la palabra int a un lado de input de la siguiente manera, si no hiciéramos esto recibimos una cadena de números y no un número.

```
>>> numero = int(input('Ingresa un número: '))
Ingresa un número: 50

>>> print(numero)
50
```

verificamos si nuestro número es una cadena o un int (entero)

```
>>> print(type(numero))
<class 'int'>
```

Platzi

Programas ramificados

Las sentencias if elif y else son palabras reservadas del lenguaje de programación python para la toma de decisiones.

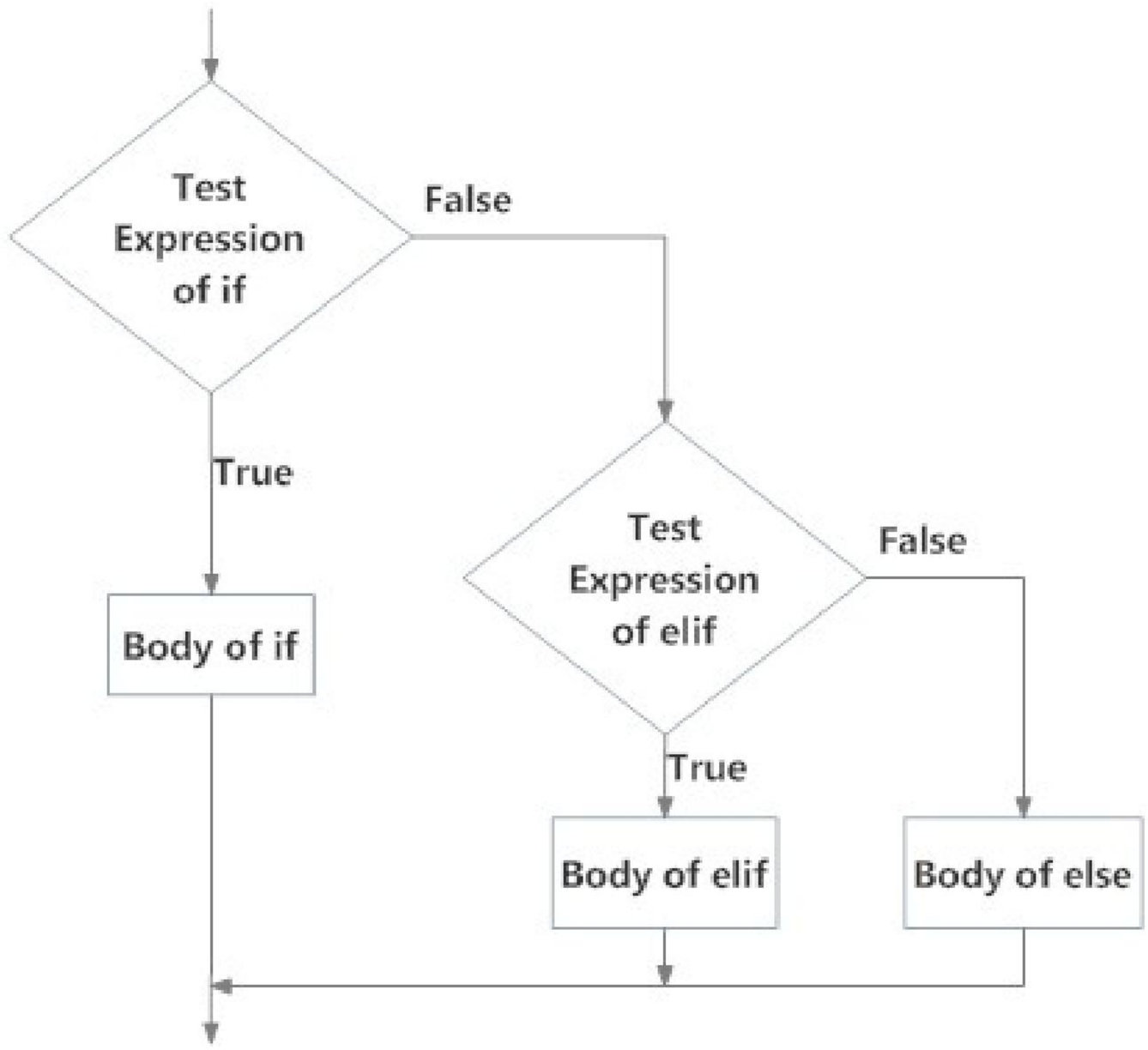


Fig: Operation of if...elif...else statement

Nuestro programa evaluará una sentencia si esta resulta ser verdadera se ejecutará el cuerpo del código.

Sentencia if

En el if evaluamos la expresión, si esta resulta ser verdadera se ejecuta el cuerpo si no, el cuerpo no se ejecuta. Pero como siempre con ejemplos es mil veces más fácil entender

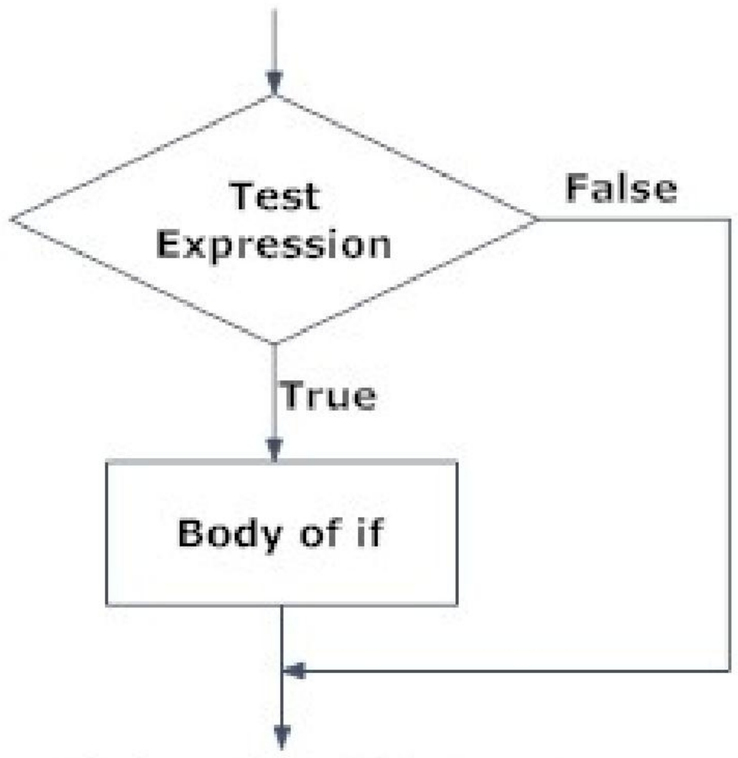


Fig: Operation of if statement

num = 3

Platzi

El siguiente programa evalúa si el número num es mayor a cero, en nuestro caso num tiene el valor de -1 por lo que la expresión es falsa y no se mostrará el mensaje “is a positive number”

```
num = -1
if num > 0:
    print(num, "is a positive number.")
print("This is always printed.")
```

Sentencia else
La sentencia else es la otra opción que se ejecuta después de que la expresión del if no fue verdadera. Por ejemplo con nuestro ejemplo anterior el num que tenía valor de -1 al no ser un número mayor a 0 no pudo imprimir que era positivo, pero si puede tener la opción que sea un número negativo y esto lo podemos imprimir con un else.

```
if num > 0:
    print("Positive number")
else:
    print("Negative number")
```

Sentencia elif
La sentencia elif se utiliza después del if, pues si la sentencia if no es verdadera puede ser que la sentencia del elif si lo sea. Si ni la sentencia del if o de los elif se ejecuta la última opción será el else . Por ejemplo

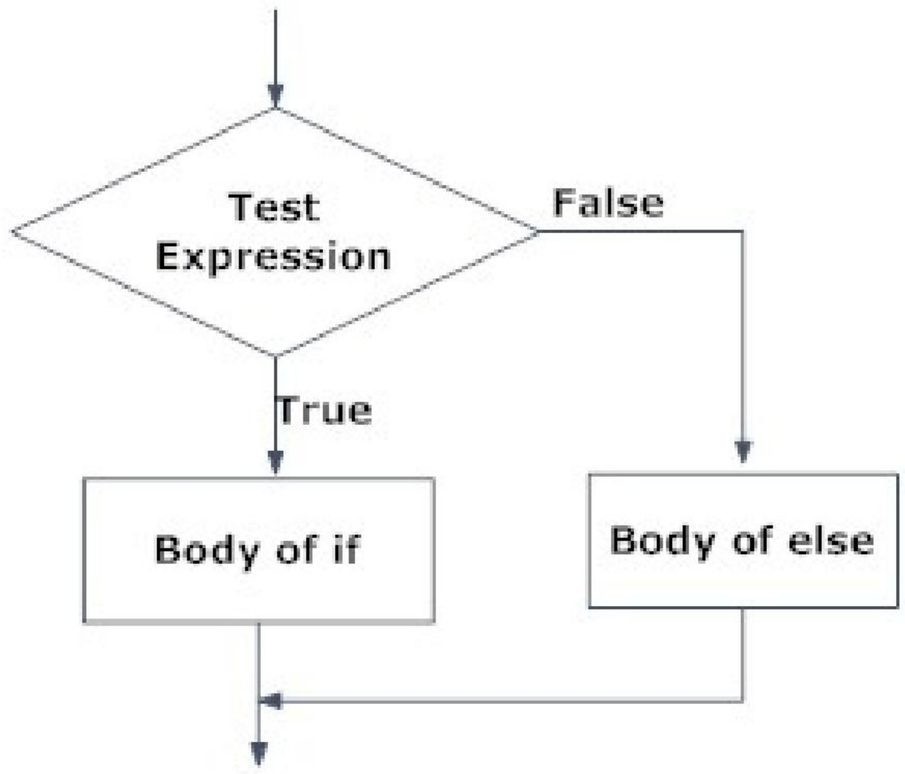


Fig: Operation of if...else statement

Platzi

```python
if num > 0:
    print("Positive number")
elif num == 0:
    print("Zero")
else:
    print("Negative number")
```

Platzi

Iteraciones

El bucle for en Python

En la definición anterior debemos entender <iterable> como una colección de objetos; y la <variable> como el elemento específico que se está exponiendo mediante el bucle en cada iteración.

```
>>> frutas = ['manzana', 'pera', 'mango']
>>> for fruta in frutas:
        print(fruta)
```

manzana
pera
mango

Iterators

Ahora que ya sabemos cómo obtener un iterator, ¿Qué podemos hacer con él? Un iterator es un objeto que regresa sucesivamente los valores asociados con el iterable.

```
>>> frutas = ['manzana', 'pera', 'mango']
>>> iterador = iter(frutas)
>>> next(iterador)
manzana
>>> next(iterador)
pera
>>> next(iterador)
mango
```

Como puedes ver, el iterator guarda el estado interno de la iteración, de tal manera que cada llamada sucesiva a next regresa el siguiente elemento. ¿Qué pasa una vez que ya no existan más elementos en el iterable? La llamada a next arrojará un error de tipo StopIteration.

Platzi

El ciclo While

```
print("""
* * * * * * * * Reloj * * * * * * * *
""")

horas = 0
minutos = 0
segundos = 0

while horas < 24:
   while minutos < 60:
       while segundos < 60:

           print("{} : {} : {}".format(horas, minutos, segundos))
           segundos += 1
       minutos += 1
       segundos = 0
   horas += 1
   minutos = 0
```

Bucles for con diccionarios

Para iterar a lo largo de un diccionario tenemos varias opciones:

- Ejecutar el bucle for directamente en el diccionario, lo cual nos permite iterar a lo largo de las llaves del diccionario.
- Ejecutar el bucle for en la llamada keys del diccionario, lo cual nos permite iterar a lo largo de las llaves del diccionario.
- Ejecutar el bucle for en la llamada values del diccionario, lo cual nos permite iterar a lo largo de los valores del diccionario.
- Ejecutar el bucle for en la llamada items del diccionario, lo cual nos permite iterar en una tupla de las llaves y los valores del diccionario.

```
estudiantes = {
    'mexico': 10,
    'colombia': 15,
    'puerto_rico': 4,
}

for pais in estudiantes:
    ...
```

Platzi

```
for pais in estudiantes.keys():
    ...

for numero_de_estudiantes in estudiantes.values():
    ...

for pais, numero_de_estudiantes in estudiantes.items():
    ...
```

Platzi

Aproximación de soluciones

Métodos para la aproximación de soluciones

La mayoría de las veces la solución de un problema *real* conduce a la elaboración de un modelo matemático, modelo que esperamos conduzca a soluciones con sentido físico. Si los modelos son sencillos, las soluciones se pueden obtener con las técnicas matemáticas normales. Sin embargo, en algunos casos es necesario recurrir a técnicas numéricas que involucran muchos cálculos y es aquí donde una herramienta como python.

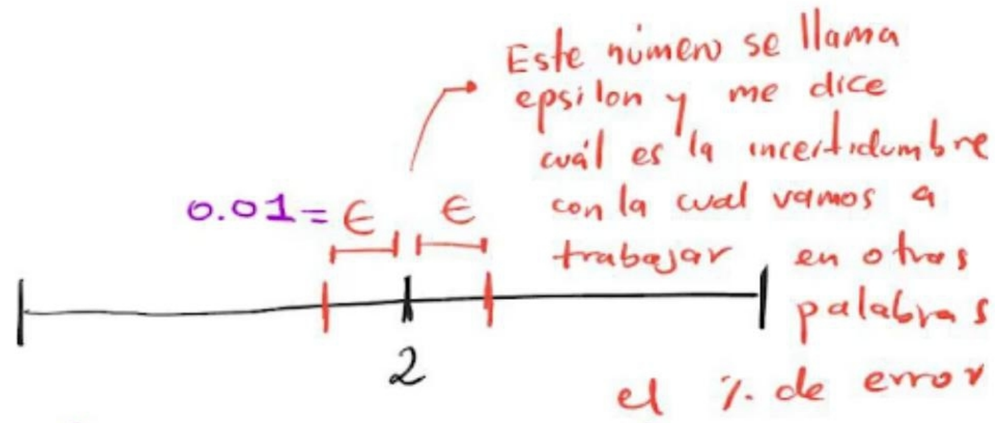
Explicación. algoritmo de aproximación.

1 - Necesitamos encontrar una posible solución a un # que el usuario nos proporcionará, [encontrar la raíz cuadrada]

supongamos que sea el → 4

2 - Por ser un cálculo que estamos familiarizados sabemos que es 2.

3 - utilizando el algoritmo solucion



¿Cuál es el error que aceptamos?

$\epsilon = 0.01 \times 100\% = 1\%$

Nota: Para el ejercicio, aceptamos que la solución encuentre un número con un % de error igual o inferior a 1%

Funciones y abstracción

Funciones:

En programación, una **función** es una sección de un programa que calcula un valor de manera independiente al resto del programa.

Una función tiene tres componentes importantes:

- los **parámetros**, que son los valores que recibe la función como entrada;
- el **código de la función**, que son las operaciones que hace la función; y
- el **resultado** (o **valor de retorno**), que es el valor final que entrega la función.

En esencia, una función es un mini programa. Sus tres componentes son análogos a la entrada, el proceso y la salida de un programa.

Abstracción:

La abstracción consiste en aislar un elemento de su contexto o del resto de los elementos que lo acompañan. En programación, el término se refiere al énfasis en el "¿qué hace?" más que en el "¿cómo lo hace?" (característica de caja negra). El común denominador en la evolución de los lenguajes de programación, desde los clásicos o imperativos hasta los orientados a objetos, ha sido el nivel de abstracción del que cada uno de ellos hace uso.

Implementando los tres métodos de solución numérica con funciones

---

```
def enumeracion_exhaustiva(objetivo):
    respuesta = 0

    while respuesta**2 < objetivo:
        respuesta += 1

    if respuesta**2 == objetivo:
        print(f'La raiz cuadrada de {objetivo} es {respuesta}')
    else:
        print(f'{objetivo} no tiene raiz exacta')
```

Platzi

```python
def busqueda_binaria(objetivo):
  epsilon = 0.001
  bajo = 0.0
  alto = max(1.0, objetivo)
  respuesta = (alto + bajo) / 2


  while abs(respuesta**2 - objetivo) >= epsilon:
    if respuesta**2 < objetivo:
      bajo = respuesta
    else:
      alto = respuesta
    respuesta = (alto + bajo) / 2

  print(f'La raiz cuadrada de {objetivo} es {respuesta}')

def aproximacion(objetivo):
  epsilon = 0.001
  paso = epsilon**2
  respuesta = 0.0

  while abs(respuesta**2 - objetivo) >= epsilon and respuesta <=
objetivo:
    respuesta += paso

  if abs(respuesta**2 - objetivo) >= epsilon:
    print(f'No se encontro la raiz cuadrada de {objetivo}')
  else:
    print(f'La raiz cuadrada de {objetivo} es {respuesta}')


def run():
  print('\n¡Bienvenido! Calculemos la raiz de un numero')
  objetivo = int(input('Escribe el numero que deseas calcular: '))
  opcion = int(input('\n 1. Enumeracion Exhaustiva \n 2. Aproximacion \n
3. Busqueda Binaria \n ¿Que opción eliges?: '))

  if opcion == 1:
    enumeracion_exhaustiva(objetivo)
  elif opcion == 2:
    aproximacion(objetivo)
  elif opcion == 3:
    busqueda_binaria(objetivo)
  else:
```

Platzi

```
    print('Ingresa una opcion valida')

if __name__ == '__main__':
  run()
```

Platzi

SCOPE

Si una variable es declarada dentro de un bloque (método/función/procedimiento), esta será válida sólo dentro de ese bloque y se destruirá al terminar el bloque. Adicionalmente, la variable no podrá verse ni usarse fuera del bloque (en el exterior del bloque). La variable dentro del bloque es una variable local y solo tiene alcance dentro del bloque que se creó y sus bloques hijos, pero no en bloques hermanos ni padres, una variable definida fuera de cualquier bloque es una variable global y cualquier bloque puede acceder a ella y modificarla.

```
def func1(un_arg, una_func):
   print("1. Entre a func1")

   def func2(otro_arg):
       print('2. Entré a func2')
       return otro_arg*2

   valor = func2(un_arg)
   print('3. valor: ',valor)
   return una_func(valor)

un_arg = 1

def cualquier_func(cualquier_arg):
   print('4. Entré a cualquier_func')
   return cualquier_arg + 5

func1(un_arg, cualquier_func)
```

Platzi

Especificaciones del código

Docstrings

En Python todos los objetos cuentan con una variable especial llamada **doc** gracias a la que podemos describir para qué sirven y cómo se usan los objetos. Estas variables reciben el nombre de *docstrings*, cadenas de documentación.

Funciones

Python implementa un sistema muy sencillo para establecer el valor de las docstrings, únicamente tenemos que crear un comentario en la primera línea después de la declaración.

Código

```
def hola(arg):
    """Este es el docstring de la función"""
    print("Hola", arg, "!")

hola("Héctor")
```

Para consultar la documentación es tan sencillo como utilizar la función reservada **help** y pasarle el objeto:

Código

```
help(hola)
```

Clases y métodos

De la misma forma podemos establecer la documentación de la clase después de la definición, y de los métodos, como si fueran funciones:

Código

```
class Clase:
    """ Este es el docstring de la clase"""
    def __init__(self):
        """Este es el docstring del inicializador de clase"""
    def metodo(self):
        """Este es el docstring del método de clase"""

o = Clase()
```

Platzi

```
help(o)
```

Scripts y módulos

Cuando tenemos un script o módulo, la primera línea del mismo hará referencia al docstring del módulo, en él deberíamos explicar el funcionamiento del mismo:

mi_modulo.py

```
"""Este es el docstring del módulo"""
def despedir():
    """Este es el docstring de la función despedir"""
    print("Adiós! Me despido desde la función despedir() del módulo
prueba")
def saludar():
    """Este es el docstring de la función saludar"""
    print("Hola! Te saludo desde la función saludar() del módulo
prueba")
```

Código

```
import mi_modulo

help(mi_modulo)
```

Código

```
help(mi_modulo.despedir)
```

Como dato curioso, también podemos listar las variables y funciones del módulo con **dir()**:

Código

```
dir(mi_modulo)
```

Como vemos muchas de ellas son especiales, seguro que muchas os suenan, os invito a comprobar sus valores:

```
Código
print(mi_modulo.__name__)     # Nombre del módulo
print(mi_modulo.__doc__)      # Docstring del módulo
print(mi_modulo.__package__)  # Nombre del paquete del módulo
```

Platzi

Paquetes

En el caso de los paquetes el docstring debemos establecerlo en la primera línea del inicializador **init**:

mi_paquete/__init__.py

```
"""Este es el docstring de mi_paquete"""
```

Platzi

Funciones como objetos

Una de las características más poderosas de Python es que todo es un objeto, incluyendo las funciones. Las funciones en Python son “ciudadanos de primera clase”.

Esto, en sentido amplio, significa que en Python las funciones:

- Tienen un tipo
- Se pueden pasar como argumentos de otras funciones
- Se pueden utilizar en expresiones
- Se pueden incluir en varias estructuras de datos (como listas, tuplas, diccionarios, etc.)

Argumentos de otras funciones

Hasta ahora hemos visto que las funciones pueden recibir parámetros para realizar los cómputos que definen. Algunos de los tipos que hemos pasado son tipos simples como cadenas, números, listas, etc. Sin embargo, también pueden recibir funciones para crear abstracciones más poderosas. Veamos un ejemplo:

```
def multiplicar_por_dos(n):
    return n * 2

def sumar_dos(n):
    return n + 2

def aplicar_operacion(f, numeros):
    resultados = []
    for numero in numeros:
        resultado = f(numero)
        resultados.append(resultado)

>>> nums = [1, 2, 3]
>>> aplicar_operacion(multiplicar_por_dos, nums)
[2, 4, 6]

>>> aplicar_operacion(sumar_dos, nums)
[3, 4, 5]
```

Platzi

Funciones en expresiones

Una forma de definir una función en una expresión es utilizando el keyword lambda. lambda tiene la siguiente sintaxis: lambda <vars>: <expresion>.

Otro ejemplo interesante es que las funciones se pueden utilizar en una expresión directamente. Esto es posible ya que como lo hemos platicado con anterioridad, en Python las variables son simplemente nombres que apuntan a un objeto (en este caso a una función). Por ejemplo:

```
sumar = lambda x, y: x + y

>>> sumar(2, 3)
5
```

Funciones en estructuras de datos

Las funciones también se pueden incluir en diversas estructuras que las permiten almacenar. Por ejemplo, una lista puede guardar diversas funciones a aplicar o un diccionario las puede almacenar como valores.

```
def aplicar_operaciones(num):
    operaciones = [abs, float]

    resultado = []
    for operacion in operaciones:
        resultado.append(operacion(num))

    return resultado

>>> aplicar_operaciones(-2)
[2, -2.0]
```

Como pudimos ver, las funciones son objetos muy versátiles que nos permiten tratarlas de diversas maneras y que nos permiten añadir capas adicionales de abstracción a nuestro programa.

Compártenos cómo te imaginas que estas capacidades de Python te pueden ayudar a escribir mejores programas.

Platzi

Código para correr en visual studio

```
def multiplicar_por_dos(n):
   return n*2

def sumar_dos(n):
   return n+2

def aplicar_operacion(f, numeros):
   resultados = []
   for numero in numeros:
       resultado = f(numero)
       resultados.append(resultado)
   return resultados

def aplicar_operaciones(num):
   operaciones = [abs, float]

   resultado = []
   for operacion in operaciones:
       resultado.append(operacion(num))

   return resultado

def run():
   nums = [1,2,3]
   res = aplicar_operacion(multiplicar_por_dos, nums)
   print(res)


   sumar = lambda x,y: x + y
   print(sumar(2,3))

   print(aplicar_operaciones(-5))

if __name__ == '__main__':
   run()
```

Platzi

Tuplas

¿Qué son las tuplas?

En Python, una **tupla** es un conjunto ordenado e inmutable de elementos del mismo o diferente tipo.

Pueden utilizarse para devolver varios valores en una función.

Las tuplas se representan escribiendo los elementos entre paréntesis y separados por comas.

```
>>> (1, "a", 3.14)
(1, 'a', 3.14)
```

En realidad no es necesario escribir los paréntesis para indicar que se trata de una tupla, basta con escribir las comas, pero Python escribe siempre los paréntesis:

```
>>> (1, "a", 3.14)
(1, 'a', 3.14)
```

La función len() devuelve el número de elementos de una tupla:

```
>>> len((1, "a", 3.14))
3
```

Una tupla puede no contener ningún elemento, es decir, ser una tupla vacía.

```
>>> ()
()
>>> len(())
0
```

Una tupla puede incluir un único elemento, pero para que Python entienda que nos estamos refiriendo a una tupla es necesario escribir al menos una coma.

Platzi

El ejemplo siguiente muestra la diferencia entre escribir o no una coma. En el primer caso Python interpreta la expresión como un número y en el segundo como una tupla de un único elemento.

```
>>> (3)
3
>>> (3,)
(3,)
```

Python escribe una coma al final en las tuplas de un único elemento para indicar que se trata de una tupla, pero esa coma no indica que hay un elemento después:

```
>>> (3,)
(3,)
>>> len((3,))
1
```

Con las tuplas también las podemos agregar, reasignandolas de la siguiente manera

```
>>>my_tuple = (1, )
>>> my_other_tuple = (2, 3, 4)
>>> my_tuple += my_other_tuple
>>> print(my_tuple)
(1, 2, 3, 4)
```

También podemos desempaquetar una tupla de la siguiente manera:

```
>>> x, y, z = my_other_tuple
>>> x
2
>>> y
3
>>> z
4
```

Regresar los valores de una función

```
>>> def coordenadas():
...   return(5,4)
>>> coordenada = coordenadas()
>>> coordenada
(5, 4)
>>> x, y = coordenadas()
>>> x
5
```

Platzi

```
>>> y
```
4

Rangos

Listas de números enteros: el tipo range()

En Python 3, range es un tipo de datos. El tipo range es una **lista inmutable de números enteros en sucesión aritmética**.

Listas para comprender range()

Las listas (list) son un tipo de datos muy flexible, que se comenta en la lección listas. Como para ver los valores de un range se necesita convertir en una lista, se comenta aquí la definición de lista, sin entrar en más detalles.

Una lista es un conjunto ordenado de elementos del mismo o diferente tipo, cuyo contenido puede modificarse.

Se representan escribiendo los elementos entre corchetes y separados por comas.

Las variables de tipo lista hacen referencia a la lista completa.

```
>>> lista = [1, "abcde", 45.5, -32]
>>> lista
[1, 'abcde', 45.5, -32]
```

Una lista que no contiene ningún elemento se denomina **lista vacía**:

```
>>> lista = [ ]
>>> lista
[ ]
```

El tipo range

El tipo range es una **lista inmutable de números enteros en sucesión aritmética**.

- Inmutable significa que, a diferencia de las listas, los range no se pueden modificar.
- Una sucesión aritmética es una sucesión en la que la diferencia entre dos términos consecutivos es siempre la misma.

Platzi

Un range se crea llamando al tipo de datos con uno, dos o tres argumentos numéricos, como si fuera una función.

Nota: En Python 2, range() se consideraba una función, pero en Python 3 no se considera una función, sino un tipo de datos , aunque se utiliza como si fuera una función.

El tipo range() con un único argumento se escribe range(n) y crea una lista inmutable de n números enteros consecutivos que empieza en 0 y acaba en n - 1.

Para ver los valores del range(), es necesario convertirlo a lista mediante la función list().

```
>>> x = range(10)
>>> x
range(0, 10)
>>> list(x)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(7)
range(0, 7)
>>> list(range(7))
[0, 1, 2, 3, 4, 5, 6]
```

Si n no es positivo, se crea un range vacío.

```
>>> list(range(-2))
[]
>>> list(range(0))
[]
```

El tipo range con dos argumentos se escribe range(m, n) y crea una lista inmutable de enteros consecutivos que empieza en m y acaba en n - 1.

```
>>> list(range(5, 10))
[5, 6, 7, 8, 9]
>>> list(range(-5, 1))
[-5, -4, -3, -2, -1, 0]
```

Si n es menor o igual que m, se crea un range vacío.

```
>>> list(range(5, 1))
[]
```

Platzi

```
>>> list(range(3, 3))
```

```
[]
```

El tipo range con tres argumentos se escribe range(m, n, p) y crea una lista inmutable de enteros que empieza en m y acaba justo antes de superar o igualar a n, aumentando los valores de p en p. Si p es negativo, los valores van disminuyendo de p en p.

```
>>> list(range(5, 21, 3))
[5, 8, 11, 14, 17, 20]
>>> list(range(10, 0, -2))
[10, 8, 6, 4, 2]
```

El valor de p no puede ser cero:

```
>>> range(4,18,0)
Traceback (most recent call last):/span>
  File "<pyshell#0>", line 1, in <module>
    range(4,18,0)
ValueError: range() arg 3 must not be zero
```

En los range(m, n, p), se pueden escribir p range distintos que generan el mismo resultado. Por ejemplo:

```
>>> list(range(10, 20, 3))
[10, 13, 16, 19]
>>> list(range(10, 21, 3))
[10, 13, 16, 19]
>>> list(range(10, 22, 3))
[10, 13, 16, 19]
```

En resumen, los tres argumentos del tipo range(m, n, p) son:

- m: el valor inicial
- n: el valor final (que no se alcanza nunca)
- p: el paso (la cantidad que se avanza cada vez).

Platzi

Si se escriben sólo dos argumentos, Python le asigna a p el valor 1. Es decir range(m, n) es lo mismo que range(m, n, 1)

Si se escribe sólo un argumento, Python, le asigna a m el valor 0 y a p el valor 1. Es decir range(n) es lo mismo que range(0, n, 1)

El tipo range() sólo admite argumentos enteros. Si se utilizan argumentos decimales, se produce un error:

```
>>> range(3.5, 10, 2)
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    range(3.5, 10, 2)
TypeError: range() integer start argument expected, got float.
```

Nota: En versiones muy antiguas de Python se podían utilizar argumentos decimales, que Python truncaba a enteros.

Concatenar range()

No se pueden concatenar tipos range(), ya que el resultado de la concatenación puede no ser un tipo range()).

```
>>> range(3) + range(5)
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    range(3) + range(5)
TypeError: unsupported operand type(s) for +: 'range' and 'range'
```

Pero sí se pueden concatenar tipos range() previamente convertidos en listas. El resultado es lógicamente una lista, que no se puede convertir a tipo range().

```
>>> list(range(3)) + list(range(5))
[0, 1, 2, 0, 1, 2, 3, 4]
```

No se pueden concatenar tipos range(), ni aunque el resultado sea una lista de números enteros en sucesión aritmética.

```
>>> range(1, 3) + range(3, 5)
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
```

Platzi

```
    range(3) + range(5)
TypeError: unsupported operand type(s) for +: 'range' and 'range'
>>> list(range(1, 3)) + list(range(3, 5))
[1, 2, 3, 4]
```

La función len()

La función len() devuelve la longitud de una cadena de caracteres o el número de elementos de una lista. El argumento de la función len() es la lista o cadena que queremos "medir".

```
>>> len("mensaje secreto")
15
>>> len(["a","b","c"])
3
>>> len(range(1, 100, 7))
15
```

El valor devuelto por la función len() se puede usar como parámetro de range().

```
>>> list(range(len("mensaje secreto")))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
>>> list(range(len(["a","b","c"])))
[0, 1, 2]
>>> list(range(len(range(1, 100, 7))))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

Platzi

Listas y mutabilidad

Listas

Las listas son una estructura de datos muy flexible. Python permite manipular listas de muchas maneras. En esta lección aprenderás algunas de ellas:

Las listas son conjuntos ordenados de elementos (números, cadenas, listas, etc). Las listas se delimitan por corchetes ([ ]) y los elementos se separan por comas.

Las listas pueden contener elementos del mismo tipo:

```
>>> primos = [2, 3, 5, 7, 11, 13]
>>> diasLaborables = ["Lunes", "Martes", "Miércoles", "Jueves",
"Viernes"]
```

O pueden contener elementos de tipos distintos:

```
>>> fecha = ["Lunes", 27, "Octubre", 1997]
```

O pueden contener listas:

```
>>> peliculas = [ ["Senderos de Gloria", 1957], ["Hannah y sus
hermanas", 1986]]
```

Las listas pueden tener muchos niveles de anidamiento:

```
>>> directores = [ ["Stanley Kubrick", ["Senderos de Gloria", 1957]],
["Woody Allen", ["Hannah y sus hermanas", 1986]] ]
```

Las variables de tipo lista hacen referencia a la lista completa.

```
>>> lista = [1, "a", 45]
>>> lista
[1, 'a', 45]
```

Al definir una lista se puede hacer referencia a otras variables.

```
>>> nombre = "Pepe"
```

Platzi

```
>>> edad = 25
>>> lista = [nombre, edad]
>>> lista
['Pepe', 25]
```

Como siempre, hay que tener cuidado al modificar una variable que se ha utilizado para definir otras variables, porque esto puede afectar al resto de variables:

Si se trata objetos inmutables, el resto de variables no resultan afectadas, como muestra el siguiente ejemplo:

```
 >>> nombre = "Pepe"
>>> edad = 25
>>> lista = [nombre, edad]
>>> lista
['Pepe', 25]
>>> nombre = "Juan"
>>> lista
['Pepe', 25]
```

Pero si se trata de objetos mutables y al modificar la variable se modifica el objeto, el resto de variables sí resultan afectadas, como muestra el siguiente ejemplo:

```
 >>> nombres = ["Ana", "Bernardo"]
>>> edades = [22, 21]
>>> lista = [nombres, edades]
>>> lista
[['Ana', 'Bernardo'], [22, 21]]
>>> nombres += ["Cristina"]
>>> lista
[['Ana', 'Bernardo', 'Cristina'], [22, 21]]
```

Una lista puede contener listas (que a su vez pueden contener listas, que a su vez etc.):

```
>>> persona1 = ["Ana", 25]
>>> persona2 = ["Benito", 23]
>>> lista = [persona1, persona2]
>>> lista
[['Ana', 25], ['Benito', 23]]
```

Platzi

Se puede acceder a cualquier elemento de una lista escribiendo el nombre de la lista y entre corchetes el número de orden en la lista. El primer elemento de la lista es el número 0.

```
>>> lista = [10, 20, 30, 40]
>>> lista[2]
30
>>> lista[0]
10
```

Se pueden concatenar dos listas utilizando la operación suma.

```
>>> lista1 = [10, 20, 30, 40]
>>> lista2 = [30, 20]
>>> lista = lista1 + lista2 + lista1
>>> lista
[10, 20, 30, 40, 30, 20, 10, 20, 30, 40]
```

Concatenar listas

Las listas se pueden concatenar con el símbolo de la suma (+):

```
>>> vocales = ["E", "I", "O"]
>>> vocales
['E', 'I', 'O']
>>> vocales = vocales + ["U"]
>>> vocales
['E', 'I', 'O', 'U']
>>> vocales = ["A"] + vocales
>>> vocales
['A', 'E', 'I', 'O', 'U']
```

El operador suma (+) necesita que los dos operandos sean listas:

```
>>> vocales = ["E", "I", "O"]
>>> vocales = vocales + "Y"
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    vocales = vocales + "Y"
  TypeError: can only concatenate list (not "str") to list
```

Platzi

También se puede utilizar el operador += para añadir elementos a una lista:

```
>>> vocales = ["A"]
>>> vocales += ["E"]
>>> vocales
['A', 'E']
```

Aunque en estos ejemplos, los operadores + y += den el mismo resultado, no son equivalentes.

Manipular elementos individuales de una lista

Cada elemento se identifica por su posición en la lista, teniendo en cuenta que se empieza a contar por 0.

```
>>> fecha = [27, "Octubre", 1997]
>>> fecha[0]
27
>>> fecha[1]
Octubre
>>> fecha[2]
1997
```

No se puede hacer referencia a elementos fuera de la lista:

```
>>> fecha = [27, "Octubre", 1997]
>>> fecha[3]
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    fecha[3]
  Index error: list index out of range
```

Se pueden utilizar números negativos (el último elemento tiene el índice -1 y los elementos anteriores tienen valores descendentes):

```
>>> fecha = [27, "Octubre", 1997]
>>> fecha[-1]
1997
```

Platzi

```
>>> fecha[-2]
Octubre
>>> fecha[-3]
27
```

Se puede modificar cualquier elemento de una lista haciendo referencia a su posición:

```
>>> fecha = [27, "Octubre", 1997]
>>> fecha[2] = 1998
>>> fecha[0]
27
>>> fecha[1]
Octubre
>>> fecha[2]
1998
```

Manipular sublistas

De una lista se pueden extraer sublistas, utilizando la notación nombreDeLista[inicio:límite], donde inicio y límite hacen el mismo papel que en el tipo range(inicio, límite).

```
>>> dias = ["Lunes", "Martes", "Miércoles", "Jueves", "Viernes",
"Sábado", "Domingo"]
>>> dias[1:4] # Se extrae una lista con los valores 1, 2 y 3
['Martes', 'Miércoles', 'Jueves']
>>> dias[4:5] # Se extrae una lista con el valor 4
['Viernes']
>>> dias[4:4] # Se extrae una lista vacía
[]
>>> dias[:4]  # Se extrae una lista hasta el valor 4 (no incluido)
['Lunes', 'Martes', 'Miércoles', 'Jueves']
>>> dias[4:]  # Se extrae una lista desde el valor 4 (incluido)
['Viernes', 'Sábado', 'Domingo']
>>> dias[:]   # Se extrae una lista con todos los valores
['Lunes', 'Martes', 'Miércoles', 'Jueves', 'Viernes', 'Sábado',
'Domingo']
```

Se puede modificar una lista modificando sublistas. De esta manera se puede modificar un elemento o varios a la vez e insertar o eliminar elementos.

```
>>> letras = ["A", "B", "C", "D", "E", "F", "G", "H"]
```

Platzi

```
>>> letras[1:4] = ["X"]      # Se sustituye la sublista ['B','C','D']
por ['X']
>>> letras
['A', 'X', 'E', 'F', 'G', 'H']
>>> letras[1:4] = ["Y", "Z"] # Se sustituye la sublista ['X','E','F']
por ['Y','Z']
['A','Y', 'Z', 'G', 'H']
>>> letras[0:1] = ["Q"]      # Se sustituye la sublista ['A'] por ['Q']
>>> letras
['Q', 'Y', 'Z', 'G', 'H']
>>> letras[3:3] = ["U", "V"] # Inserta la lista ['U','V'] en la posición
3
>>> letras
['Q', 'Y', 'Z', 'U', 'V', 'G', 'H']
>>> letras[0:3] = []         # Elimina la sublista ['Q','Y', 'Z']
>>> letras
['U', 'V', 'G', 'H']
```

Al definir sublistas, Python acepta valores fuera del rango, que se interpretan como extremos (al final o al principio de la lista).

```
>>> letras = ["D", "E", "F"]
>>> letras[3:3] = ["G", "H"]          # Añade ["G", "H"] al final de la
lista
>>> letras
['D', 'E', 'F', 'G', 'H']
>>> letras[100:100] = ["I", "J"]      # Añade ["I", "J"] al final de la
lista
>>> letras
['D', 'E', 'F', 'G', 'H', 'I', 'J']
>>> letras[-100:-50] = ["A", "B", "C"] # Añade ["A", "B", "C"] al
principio de la lista
>>> letras
['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J']
```

La palabra reservada del

La palabra reservada del permite eliminar un elemento o varios elementos a la vez de una lista, e incluso la misma lista.

```
>>> letras = ["A", "B", "C", "D", "E", "F", "G", "H"]
>>> del letras[4]   # Elimina la sublista ['E']
>>> letras
```

Platzi

```
['A', 'B', 'C', 'D', 'F', 'G', 'H']
>>> del letras[1:4] # Elimina la sublista ['B', 'C', 'D']
>>> letras
['A', 'F', 'G', 'H']
>>> del letras      # Elimina completamente la lista
>>> letras
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in ?
    letras
  NameError: name 'letras' is not defined
```

Si se intenta borrar un elemento que no existe, se produce un error:

```
>>> letras = ["A", "B", "C", "D", "E", "F", "G", "H"]
>>> del letras[10]
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    del letras[10]
IndexError: list assignment index out of range
```

Aunque si se hace referencia a sublistas, Python sí que acepta valores fuera de rango, pero lógicamente no se modifican las listas.

```
>>> letras = ["A", "B", "C", "D", "E", "F", "G", "H"]
>>> del letras[100:200] # No elimina nada
>>> letras
['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H']
```

Copiar una lista

Con variables de tipo entero, decimal o de cadena, es fácil tener una copia de una variable para conservar un valor que en la variable original se ha perdido:

```
>>> a = 5
>>> b = a # Hacemos una copia del valor de a
>>> a, b
(5, 5)
>>> a = 4 # de manera que aunque cambiemos el valor de a ...
>>> a, b  # ... b conserva el valor anterior de a en caso de necesitarlo
(4, 5)
```

Platzi

Pero si hacemos esto mismo con listas, nos podemos llevar una sorpresa:

```
>>> lista1 = ["A", "B", "C"]
>>> lista2 = lista1 # Intentamos hacer una copia de la lista lista1
>>> lista1, lista2
(['A', 'B', ,'C'] ['A', 'B', ,'C'])
>>> del lista1[1]   # Eliminamos el elemento ['B'] de la lista lista1
...
>>> lista1, lista2  # ... pero descubrimos que también ha desaparecido
de la lista lista2
(['A', 'C'] ['A', 'C'])
```

El motivo de este comportamiento, es que los enteros, decimales y cadenas son objetos inmutables y las listas son objetos mutables.

Si queremos copiar una lista, de manera que conservemos su valor aunque modifiquemos la lista original debemos utilizar la notación de sublistas.

```
>>> lista1 = ["A", "B", "C"]
>>> lista2 = lista1[:] # Hacemos una copia de la lista lista1
>>> lista1, lista2
(['A', 'B', 'C'] ['A', 'B', 'C'])
>>> del lista1[1]      # Eliminamos el elemento ['B'] de la lista lista1
...
>>> lista1, lista2     # ... y en este caso lista2 sigue conservando el
valor original de lista1
(['A', 'C'] ['A', 'B', 'C'])
```

En el primer caso las variables lista1 y lista2 hacen referencia a la misma lista almacenada en la memoria del ordenador. Por eso al eliminar un elemento de lista1, también desaparece de lista2.

Sin embargo en el segundo caso lista1 y lista2 hacen referencia a listas distintas (aunque tengan los mismos valores, están almacenadas en lugares distintos de la memoria del ordenador). Por eso, al eliminar un elemento de lista1, no se elimina en lista2.

Platzi

A continuación se ejecuta la segunda instrucción del bloque.

Esta instrucción escribe la lista letras, que ha perdido el elemento "B".

Saber si un valor está o no en una lista

Para saber si un valor está en una lista se puede utilizar el operador in. La sintaxis sería "elemento in lista" y devuelve un valor lógico: True si el elemento está en la lista, False si el elemento **no** está en la lista.

Por ejemplo, el programa siguiente comprueba si el usuario es una persona autorizada:

```
personas_autorizadas = ["Alberto", "Carmen"]
nombre = input("Dígame su nombre: ")
if nombre in personas_autorizadas:
  print("Está autorizado")
else:
  print("No está autorizado")
```

Para saber si un valor **no** está en una lista se pueden utilizar los operadores not in. La sintaxis sería "elemento not in lista" y devuelve un valor lógico: True si el elemento **no** está en la lista, False si el elemento está en la lista.

Por ejemplo, el programa siguiente comprueba si el usuario es una persona autorizada:

```
personas_autorizadas = ["Alberto", "Carmen"]
nombre = input("Dígame su nombre: ")
if nombre not in personas_autorizadas:
  print("No está autorizado")
else:
  print("Está autorizado")
```

Platzi

List comprehension

- Es una forma concisa de aplicar operaciones a los valores de una secuencia.
- También se pueden aplicar condiciones para filtrar

Vamos a realizar una serie de operaciones con una lista

```
>>> my_list = list(range(100))
>>> my_list
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37,
38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55,
56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73,
74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91,
92, 93, 94, 95, 96, 97, 98, 99]
```

De esta lista de números multiplicamos cada uno de ellos utilizando list comprehension.

```
>>> double = [i * 2 for i in my_list]
>>> double
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36,
38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72,
74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 94, 96, 98, 100, 102, 104, 106,
108, 110, 112, 114, 116, 118, 120, 122, 124, 126, 128, 130, 132, 134,
136, 138, 140, 142, 144, 146, 148, 150, 152, 154, 156, 158, 160, 162,
164, 166, 168, 170, 172, 174, 176, 178, 180, 182, 184, 186, 188, 190,
192, 194, 196, 198]
```

Haremos un programa para imprimir los números pares del 0 al 99

```
>>> pares = [i for i in my_list if i % 2 == 0]
>>> pares
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36,
38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72,
74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 94, 96, 98]
```

Platzi

Diccionarios

¿Qué es un diccionario de datos?

Un Diccionario es una estructura de datos y un tipo de dato en Python con características especiales que nos permite almacenar cualquier tipo de valor como enteros, cadenas, listas e incluso otras funciones. Estos diccionarios nos permiten además identificar cada elemento por una clave (Key).

Para definir un diccionario, se encierra el listado de valores entre llaves. Las parejas de clave y valor se separan con comas, y la clave y el valor se separan con dos puntos.

```
diccionario = {'nombre' : 'Carlos', 'edad' : 22, 'cursos':
['Python','Django','JavaScript'] }
```

1.

Podemos acceder al elemento de un Diccionario mediante la clave de este elemento, como veremos a continuación:

```
print diccionario['nombre'] #Carlos
print diccionario['edad']#22
print diccionario['cursos'] #['Python','Django','JavaScript']
```

2.

También es posible insertar una lista dentro de un diccionario. Para acceder a cada uno de los cursos usamos los índices:

```
print diccionario['cursos'][0]#Python
print diccionario['cursos'][1]#Django
print diccionario['cursos'][2]#JavaScript
```

3.

Para recorrer todo el Diccionario, podemos hacer uso de la estructura for:

```
for key in diccionario:
  print (key, ":", diccionario[key])
```

Platzi

Métodos de los Diccionarios

dict ()
Recibe como parámetro una representación de un diccionario y si es factible, devuelve un diccionario de datos.

```
dic =  dict(nombre='nestor', apellido='Plasencia', edad=22)

dic → {'nombre' : 'nestor', 'apellido' : 'Plasencia', 'edad' : 22}
```

zip()
Recibe como parámetro dos elementos iterables, ya sea una cadena, una lista o una tupla. Ambos parámetros deben tener el mismo número de elementos. Se devolverá un diccionario relacionando el elemento i-esimo de cada uno de los iterables.

```
dic = dict(zip('abcd',[1,2,3,4]))

dic →   {'a' : 1, 'b' : 2, 'c' : 3 , 'd' : 4}
```

items()
Devuelve una lista de tuplas, cada tupla se compone de dos elementos: el primero será la clave y el segundo, su valor.

```
dic = dict(zip('abcd',[1,2,3,4]))

dic →   {'a' : 1, 'b' : 2, 'c' : 3 , 'd' : 4}
```

keys()
Retorna una lista de elementos, los cuales serán las claves de nuestro diccionario.

```
dic =  {'a' : 1, 'b' : 2, 'c' : 3 , 'd' : 4}
keys= dic.keys()

keys→ ['a','b','c','d']
```

values()
Retorna una lista de elementos, que serán los valores de nuestro diccionario.

```
dic =  {'a' : 1, 'b' : 2, 'c' : 3 , 'd' : 4}
values= dic.values()

values→ [1,2,3,4]
```

Platzi

clear()
Elimina todos los ítems del diccionario dejándolo vacío.

```
dic 1 =  {'a' : 1, 'b' : 2, 'c' : 3 , 'd' : 4}
dic1.clean()

dic1 → { }
```

copy()
Retorna una copia del diccionario original.

```
dic = {'a' : 1, 'b' : 2, 'c' : 3 , 'd' : 4}
dic1 = dic.copy()

dic1 → {'a' : 1, 'b' : 2, 'c' : 3 , 'd' : 4}
```

fromkeys()
Recibe como parámetros un iterable y un valor, devolviendo un diccionario que contiene como claves los elementos del iterable con el mismo valor ingresado. Si el valor no es ingresado, devolverá none para todas las claves.

```
dic = dict.fromkeys(['a','b','c','d'],1)

dic →  {'a' : 1, 'b' : 1, 'c' : 1 , 'd' : 1}
```

get()
Recibe como parámetro una clave, devuelve el valor de la clave. Si no lo encuentra, devuelve un objeto none.

```
dic = {'a' : 1, 'b' : 2, 'c' : 3 , 'd' : 4}
valor = dic.get('b')

valor → 2
```

pop()
Recibe como parámetro una clave, elimina esta y devuelve su valor. Si no lo encuentra, devuelve error.

```
dic = {'a' : 1, 'b' : 2, 'c' : 3 , 'd' : 4}
valor = dic.pop('b')

valor → 2
```

Platzi

```
dic → {'a' : 1, 'c' : 3 , 'd' : 4}
```

setdefault()
Funciona de dos formas. En la primera como get

```
dic = {'a' : 1, 'b' : 2, 'c' : 3 , 'd' : 4}
valor = dic.setdefault('a')

valor → 1
```

Y en la segunda forma, nos sirve para agregar un nuevo elemento a nuestro diccionario.

```
dic = {'a' : 1, 'b' : 2, 'c' : 3 , 'd' : 4}
valor = dic.setdefault('e',5)

dic → {'a' : 1, 'b' : 2, 'c' : 3 , 'd' : 4 , 'e' : 5}
```

update()
Recibe como parámetro otro diccionario. Si se tienen claves iguales, actualiza el valor de la clave repetida; si no hay claves iguales, este par clave-valor es agregado al diccionario.

```
dic 1 = {'a' : 1, 'b' : 2, 'c' : 3 , 'd' : 4}
dic 2 = {'c' : 6, 'b' : 5, 'e' : 9 , 'f' : 10}
dic1.update(dic 2)
```

```
dic 1 → {'a' : 1, 'b' : 5, 'c' : 6 , 'd' : 4 , 'e' : 9 , 'f' : 10}
```

Estos son algunos de los métodos más útiles y más utilizados en los Diccionarios. Python es un gran lenguaje de programación que nos permite programar de una manera realmente sencilla.

Dictionary Comprehension

Un ejemplo de Dictionary Comprehension 🐍

Este programa utiliza comprehension para multiplicar por dos todos los valores del diccionario.

```
dict1 = {'a': 1, 'b': 2, 'c': 3, 'd': 4}
double_dict1 = {key:values*2 for (key, values) in dict1.items()}
print(double_dict1)
```

Platzi

Pruebas de caja negra

Nos ayuda a asegurar el comportamiento que esperamos y encontrar de manera eficiente y sistemática los errores dentro de nuestro código.

- Se basan en la especificación de la función o el programa
- Prueba inputs y valida outputs
- Unit testing o integration testing

```
import unittest

def suma(num_1, num_2):
   return abs(num_1) + num_2

class caja_negra_test(unittest.TestCase):

   def test_suma_dos_positivos(self):
       num_1 = 10
       num_2 = 5

       resultado = suma(num_1, num_2)

       self.assertEqual(resultado, 15)

   def test_suma_dos_negativos(self):
       num_1 = -10
       num_2 = -7

       resultado = suma(num_1, num_2)

       self.assertEqual(resultado, -17)


if __name__ == '__main__':
   unittest.main()
```

Descripción del código: En el código anterior hacemos pruebas con la librería unittest de python para probar las dos funciones que están dentro de nuestra caja negra que en este caso es una función que suma dos positivos y otra que suma dos negativos el objetivo es identificar si se tienen los valores esperados sabemos que al ejecutar el código uno fallará pues en la función de suma retornamos la suma de un valor absoluto, para el caso de la suma de los dos positivos no hay problema pero en el caso de los dos negativos si.

Platzi

Pruebas de caja de cristal o caja blanca

Las pruebas de caja blanca (también conocidas como pruebas de caja de cristal o pruebas estructurales) se centran en los detalles procedimentales del software, por lo que su diseño está fuertemente ligado al código fuente. El ingeniero de pruebas escoge distintos valores de entrada para examinar cada uno de los posibles flujos de ejecución del programa y cerciorarse de que se devuelven los valores de salida adecuados.

Aunque las pruebas de caja blanca son aplicables a varios niveles —unidad, integración y sistema—, habitualmente se aplican a las unidades de software. Su cometido es comprobar los flujos de ejecución dentro de cada unidad (función, clase, módulo, etc.) pero también pueden probar los flujos entre unidades durante la integración, e incluso entre subsistemas, durante las pruebas de sistema.

Las principales técnicas de diseño de pruebas de caja blanca son:

- Pruebas de flujo de control
- Pruebas de flujo de datos
- Pruebas de bifurcación (*branch testing*)
- Pruebas de caminos básicos

Basic procedure

White-box testing's basic procedures require the tester to have an in-depth knowledge of the source code being tested. The programmer must have a deep understanding of the application to know what kinds of test cases to create so that every visible path is exercised for testing. Once the source code is understood then the source code can be analyzed for test cases to be created. The following are the three basic steps that white-box testing takes in order to create test cases:

1. Input involves different types of requirements, functional specifications, detailed designing of documents, proper source code and security specifications. This is the preparation stage of white-box testing to lay out all of the basic information.
2. Processing involves performing risk analysis to guide whole testing process, proper test plan, execute test cases and communicate results. This is the phase of building test cases to make sure they thoroughly test the application the given results are recorded accordingly.
3. Output involves preparing final report that encompasses all of the above preparations and results.

Platzi

Debugging

In computer programming and software development, **debugging** is the process of finding and resolving *bugs* (defects or problems that prevent correct operation) within computer programs, software, or systems.

Debugging tactics can involve interactive debugging, control flow analysis, unit testing, integration testing, log file analysis, monitoring at the application or system level, memory dumps, and profiling. Many programming languages and software development tools also offer programs to aid in debugging, known as *debuggers*.

Techniques

- *Interactive debugging*
- *Print debugging* (or tracing) is the act of watching (live or recorded) trace statements, or print statements, that indicate the flow of execution of a process. This is sometimes called *printf debugging*, due to the use of the printf function in C. This kind of debugging was turned on by the command TRON in the original versions of the novice-oriented BASIC programming language. TRON stood for, "Trace On." TRON caused the line numbers of each BASIC command line to print as the program ran.

Reglas generales

- No te molestes con el debugger. Aprende a utilizar el print statement
- Estudia los datos disponibles
- Utiliza los datos para crear hipótesis y experimentos . Método científico
- Ten una mente abierta. Si entendieras el programa probablemente no habrían bugs
- Lleva un registro de lo que has tratado, preferentemente en la forma de tests

Diseño de experimentos

- Debuggear es un proceso de búsqueda. Cada prueba debe acotar el espacio de búsqueda.
- Búsqueda binaria por print statements

Errores comunes

- Encuentra los sospechosos comunes (errores de typing, comparar flotantes, uso de funciones como append)
- En lugar de preguntarte por qué un programa no funciona, pregúntate por qué está funcionando de esta manera
- Es posible que el bug no se encuentre donde crees que está

Platzi

- Explicale el programa a otra persona, de preferencia que no tenga contexto
- Lleva un registro de lo que has tratado, preferentemente en la forma de tests
- Vete a dormir.

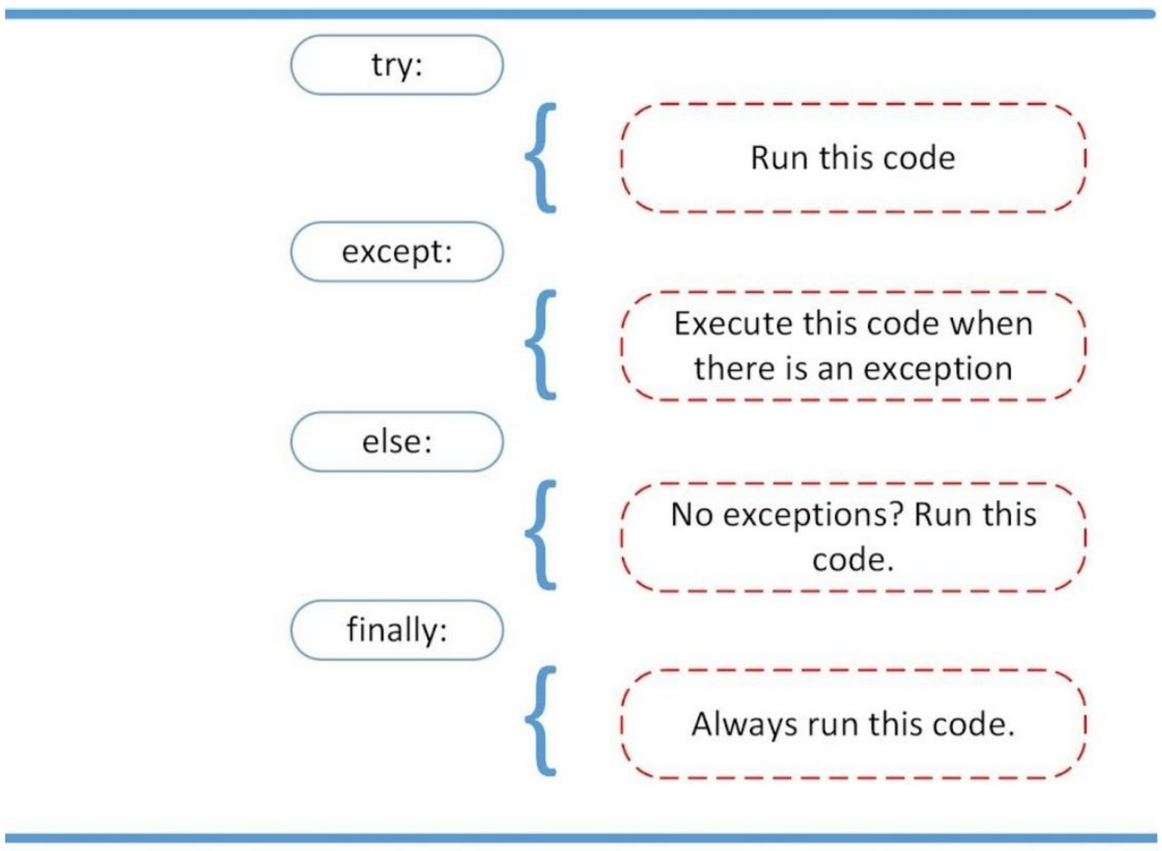
Platzi

Manejo de excepciones

- Son muy comunes en la programación. No tienen nada excepcional
- Las excepciones de Python normalmente se relacionan con errores de semántica
- Se pueden crear excepciones propias
- Cuando una excepción no se maneja (unhandled exception), el programa termina en error

Excepciones:

- Las excepciones se manejan con los keywords: try, except, finally
- Se pueden utilizar también para ramificar programas.
- No deben manejarse de manera silenciosa (por ejemplo, con print statements)
- Para aventar tu propia excepción utiliza el keyword raise.



Excepciones comunes:

ImportError : una importación falla;
IndexError : una lista se indexa con un número fuera de rango;
NameError : se usa una variable desconocida ;
SyntaxError : el código no se puede analizar correctamente
TypeError : se llama a una función en un valor de un tipo inapropiado;
ValueError : se llama a una función en un valor del tipo correcto, pero con un valor inapropiado

Platzi

Excepciones como control de flujo

Hasta ahora hemos visto como las excepciones nos permiten controlar los posibles errores que pueden ocurrir en nuestro código. Sin embargo, dentro de la comunidad de Python tienen otro uso: control de flujo.

En este momento ya debes estar familiarizado con las estructuras de control flujo que ofrece Python (if... elif...else); entonces, ¿por qué es necesaria otra modalidad para controlar el flujo? Una razón muy específica: el principio EAFP (*easier to ask for forgiveness than permission*, es más fácil pedir perdón que permiso, por sus siglas en inglés).

El principio EAFP es un estilo de programación común en Python en el cual se asumen llaves, índices o atributos válidos y se captura la excepción si la suposición resulta ser falsa. Es importante resaltar que otros lenguajes de

programación favorecen el principio LBYL (*look before you leap*, revisa antes de saltar) en el cual el código verifica de manera explícita las precondiciones antes de realizar llamadas.

Veamos ambos estilos:

Python

```
def busca_pais(paises, pais):
    """
    Paises es un diccionario. Pais es la llave.
    Codigo con el principio EAFP.
    """

    try:
        return paises[pais]
    except KeyError:
        return None
```

// Javascript

```
/**
* Paises es un objeto. Pais es la llave.
* Codigo con el principio LBYL.
*/
function buscaPais(paises, pais) {
  if(!Object.keys(paises).includes(pais)) {
    return null;
  }

  return paises[pais];
}
```

Platzi

Como puedes ver, el código de Python accede directamente a la llave y únicamente si dicho acceso falla, entonces se captura la excepción y se provee el código necesario. En el caso de JavaScript, se verifica primero que la llave exista en el objeto y únicamente con posterioridad se accede.

Es importante resaltar que ambos estilos pueden utilizarse en Python, pero el estilo EAFP es mucho más “pythonico”.

Script completo

```
def busca_pais(paises, pais):
    """
    Paises es un diccionario. País es la llave.
    Código con el principio EAFP.
    """
    try:
        return paises[pais]
    except KeyError:
        return None

def run():
    paises = {
        'Mexico':1,
        'Argentina':2,
        'Colombia':3
    }

    pais = input('Ingrese su pais: ')

    print(busca_pais(paises, pais))


if __name__ == '__main__':
    run()
```

Platzi

Afirmaciones

Utilizamos la palabra reservada assert que es usada durante el debugging (búsqueda de errores) y de forma defensiva.
El keyword assert nos permite probar si una condición en nuestro código devuelve True, si no es así, el programa mostrará un Assertion Error.
Nosotros podemos escribir el código a ejecutar en caso de que se retorne un falso.

- Programación defensiva
- Pueden utilizarse para verificar que los tipos sean correctos en una función.
- También sirven para debuggear.

aporte de @Angelo Paul Yenque Tume

```
def first_word(list_words):
    first_words = []

    for word in list_words:
        try:
            assert type(word) == str, f'{word} is not a string'
            assert len(word) > 0,'Empty strings are not allowed'
            first_words.append(word)
        except AssertionError as e:
            print(e)

    return first_words

def run():
    My_list = ['Angelo',5.5, '', 2 , '43952353', 0.35]
    print('First allowed strings: ' , first_word(My_list))

if __name__ == '__main__':
    run()
```

Platzi

Código afirmaciones

```
def primera_letra(lista_de_palabras):
    primeras_letras = []

    for palabra in lista_de_palabras:

        assert type(palabra)==str, f'{palabra} no es str'
        assert len(palabra) > 0, 'No se permiten str vacios'

        primeras_letras.append(palabra[0])

    return primeras_letras



def run():
    lista_de_palabras = ['Aldo', '', 0, '345', '', 3.5]
    print(primera_letra(lista_de_palabras))

if __name__ == '__main__':
    run()
```

Platzi