# SCRUM

Product Owner -> Persona que está del lado del cliente. Se encarga de conocer las necesidades del cliente y posteriormente lo transmite al grupo.

Scrum Master (Líder)(Moderador) -> Se encarga de dar órdenes. Va a ayudar a entender al grupo la necesidad del cliente y solucionará problemas.

Development Team (Equipo de desarrollo) -> Personas capacitadas para construir la necesidad de los clientes.

El product owner define ARTEFACTO, documente que tiene la lista completa de necesidades. Ese artefacto se llama PRODUCT BACKLOG. El product owner mediante SPRING PLANNING MEETING informará al development team de las necesidades. De esa reunión se sacará el SPRING BACKLOG que serán los objetivos que se deberán lograr entre 1-4 semanas. Ese tiempo es llamado SPRINT. El sprint es un proceso de desarrollo de la necesidad del cliente. En el sprint intervienen el Scrum Master y el Development Team.

Mediante el Daily Scrum se hará un seguimiento diario de los procesos de sprint.

1. ¿Qué se hizo ayer?
2. ¿Qué se hará hoy?
3. ¿Qué se hará mañana?
4. ¿Qué problemas se han tenido?

Se suele hacer una tabla donde se ponen los objetivos planteados, los objetivos en proceso y los objetivos terminados.

Con el Sprint Review se revisa el producto

# Con el Sprint Retrospective se analizan los resultados del sprint para encontrar errores y corregirlos.

# EL PODER DEL NOMBRE DE LAS VARIABLES

1. Hay que poner unos buenos nombres a las variables, apropiado, fácil de recordar. Y tiene que describir aquello que representa.
2. Los nombres cortos no suelen ser una buena opción, lo óptimo es que contengan entre 10 y 16 caracteres.

Los nombres “largos” están mejor usados en variables globales, fácil de recordar. Y los nombres “cortos” están mejor usados en variables locales, donde se usarán pocas veces.

1. Cuando ponemos modificadores en los nombres de las variables, hay que ponerlo al final del nombre (Total, Sum, Max, Min, String…)
2. Las letras “i”, “j” y “k” se suelen usar solamente para bucles simples.
3. Para los variables booleanas es correcto la utilización de “Done”, “found”(un valor encontrado) y “success” o “ok”.

Y no es correcto usar nombres negativos (notFound, notDone…), es mejor usar solamente nombres positivos.

1. Los prefijos que se ponen a los nombres tienen que ponerse detrás de los nombres.

# BUENOS NOMBRES A LAS RUTINAS

1. Un buen nombre a una rutina describe aquello que hace o resultado.

Con sentido y no impreciso

1. Haz los nombres de las rutinas lo largo que sea necesario, entre 9 y 15 palabras.
2. Para nombrar a una función es necesario un verbo fuerte seguido de un objeto.

# CONDICIONALES

Los condicionales controlan la ejecución de otro esdevenimiento.

* if
* else
* case
* switch

# 

# ITERACIONES/BUCLES

* for
* while
* do-while

Para salir de un bucle tenemos “break” y para continuar el bucle tenemos “continue”

## WHILE

Cuando no conocemos el número de veces que queremos hacer el bucle

## FOR

Cuando conocemos el número de veces que se tiene que repetir el bucle.

# GIT FLOW

## BRANCHES

git branch “nombre de la rama” → Crear rama

git checkout “nombre de la rama” → Cambiar de rama

git merge “rama” → Combina la rama llamada con la actual

git branch -d “rama” → Eliminar una rama

## CREAR REPOSITORIOS

git init → convertir un directorio existente a repositorio

git clone “URL” → Descarga el repo de GitHub

## GIT IGNORE

Todos los archivos que se encuentren dentro de GitIgnore, no se subirán a GitHub.

# SEMANTIC VERSIONING

Proceso de asignar nombres o números únicos a un determinado estado de un software.

SemVer (Semantic Versioning) -- 1.0.2

1 → Mayor (cambios en la api)

0 → Minor (cambios en la funcionalidad)(no rompen nada)

2 → Patch (arreglo de bugs)(no rompen api)

ComVer (Compatible Versioning) -- 1.0

## PLACE HOLDERS

* Signo de intercalación (^1.2.3) → Actualizaciones en el MINOR Y PATCH
* Virguililla (~1.2.3) → Actualizaciones en el MINOR (1.2.4, 1.2.5), no (1.3)
* Rango (>= 1.2.3, <2.0.0) → Actualizaciones entre la primera versión y la segunda
* Versión exacta (1.2.3) → Actualización en esa versión
* Place Holders X (1.X) → Todas las versiones del 1.

# DEBBUGGING

# VIRTUAL ENV

1. Primero de todo tendríamos que tener python.
2. Creamos un directorio para clonar o crear el repositorio
3. Clonamos el repo
4. Iniciamos el entorno virtual para instalar las dependencias solamente allí

# UDACITY

LESSON 1 UDACITY

Un polimorfismo es cuando un STRING y un ENTERO se comportan de la misma manera.

Los objetos “inmutables” son los OBJETOS y los STRINGS.

<no terminal> -> Puede ser reemplazado por otro <no terminal> o por un <terminal>.

Derivación -> Empezar con <nt> y acabar con <t>

Subject <nt> Verb <nt> Object <nt>

Noun <nt> Verb <nt> Object <nt>

I <t> Like <t> Noun <nt>

Python <t>

I like Python

Un compilador es un programa que genera otros programas (python).

Hace todo el trabajo de una vez y luego ejecuta el nuevo programa.

Python -> Hace el trabajo al mismo tiempo.

Un string y un entero no se pueden sumar, pero se pueden multiplicar:

print “Hello” + name(Carlos) + “!” \* 3

Hello Carlos!!!

INDEXING STRINGS

“udacity” [0] -> “u”

“udacity” [1+1] -> “a”

“udacity” [0:3] -> “uda”

“udacity” [-1] -> “y”

“udacity” [3:] -> “city”

“udacity” [:4] -> “udac”

“udacity” [:] -> “udacity”

FIND STRINGS IN STRINGS

<string>.find(<string>)

Si lo que se está buscando no existe = -1

<string>.find(<string>, <number>)

LESSON 5. UDACITY

Procedimientos -> Forma de empaquetar código para poder utilizarlo con más facilidad

Control -> Forma de ser capaces de tomar decisiones y de repetir acciones

def -> Para definir procedimientos

def <name> (<parametros> (inputs))

<block> -> Secuencia de sentencias (código que se ejecuta al ejecutar el procedimiento)

Ejemplo:

def get\_next\_tarjet(s):

start\_link = s.find(“<a href=”)

start\_quote = s.find(“””, start\_link)

end\_quote = s.find (“””, start\_quote + 1)

url = s[start\_quote + 1 : end\_quote]

return url, end\_quote

USANDO PROCEDIMIENTOS

<procedimiento> (<input>, <input>...)

Ejemplo:

def sum(a,b):

a = a + b

return a

print sum(2,123) -> 125

Ejemplo2:

def find\_second(tarjet):

first = search.find(tarjet)

second = serch.find(tarjet, first + 1)

return second

print find\_second(danton, “audace”)

==

def find\_second(search, tarjet):

return search.find(target, search.find(target + 1)

print find\_second(danton, “audace”)

VALORES DE COMPARACIÓN

<, >, !=, ==

IF

if <testexpression>:

<block>

Ejemplo:

def bigger(a,b):

if a > b:

return a

return b

print bigger(1,5) -> 5

==

def bigger(a,b):

if a > b:

return a

else:

return b

print bigger(1, 5) -> 5

OR

def is\_friend(name):

if name[0] == “D” or name[0] == “N”

BUCLES

WHILE

while <testexpression>:

<block>

Ejemplo:

def print\_numbers(n):

i = 1

while i <= n:

print i

i = i + n

==

def print\_number(10):

i = 0

while i <= 10:

print i

i = i + 1

FACTORIAL

factorial (n) = n \* (n-1) \* (n-2) \* (n-3)... \*1

Ejemplo:

def factorial(n):

result = 1

while n >= 1:

result = result \* n

n = n - 1

return result

print factorial(n)

BREAK

while <testexpression>:

<code>

if <breaktest>:

break

CÓMO RESOLVER PROBLEMAS

Para resolver un problemas, necesitamos entenderlo.

¿Cuáles son las entradas?

“Dantu tu cumpleaños y la fecha actual, calcula tu edad en días”

2 fechas -> Inputs

def daysBetweenDays(year1, month1, day1, year2, month2, day2):

¿Cuáles son las salidas?

“Dantu tu cumpleaños y la fecha actual, calcula tu edad en días”

“return” un número de días entre la primera y segunda fecha.

¿Cuáles son los resultados? // Solucionar el problema

|  |
| --- |
| daysBetweenDays(2013, 1, 24, 2013, 6, 29)  defNextDay(year, month, day):  if day < 30:  return year, month, day + 1  else:  if month == 12:  return year + 1, 1, 1  else:  return year, month + 1, 1  def dateIsBefore(year1, month1, day1, year2, month2, day2):  if year1 < year2:  return True  if year1 == year2:  if month1 < month2:  return True  if month1 == month2:  return day1 < day2  return False  def daysBetweenDates(year1, month1, day1, year2, month2, day2):  days = 0  while dateIsBefore(year1, month1, day1, year2, month2, day2):  return year1, month1, day1 = nextDay(year1, month1, day1)  days += 1  return days |

Para poner los años bisiestos y que tengan los días del mes que le corresponde.

|  |
| --- |
| def isLeapYear(year):  if year % 400 == 0:  return True  if year % 100 == 0:  return False  if year % 4 == 0:  return True  return False  def daysInMonth(year, month):  if month == 1 or month == 3 or month == 5 or month == 8 or month == 10 /  or month == 12:  return 31  if month == 2:  if isLeapYear(year):  return 29  return 28  else:  return 30  def nextDay (year, month, day):  if day < daysInMonth(year, month)  return year, month, day + 1  else:  if month == 12:  return year + 1, 1, 1  else:  return year, month + 1, 1  def dateIsBefore(year1, month1, day1, year2, month2, day2):  if year1 < year2:  return True  if year1 == year2:  if month1 < month2:  return True  if month1 == month2:  return day1 < day2  return False  def daysBetweenDates(year1, month1, day1, year2, month2, day2):  days = 0  while dateIsBefore(year1, month1, day1, year2, month2, day2):  return year1, month1, day1 = nextDay(year1, month1, day1)  days += 1  return days |

LISTAS

<list> -> [<expression>, <expression>...]

|  |
| --- |
| daysInMonth = [31,28,31,30,31,30,31,31,30,31,30,31]  def how\_many\_days(month\_number):  return daysInMonth[month\_number - 1]  print how\_many\_days(6) -> 30 |

|  |
| --- |
| countries = [[“China”, “Beijing”, 1350], [“India”, “Delhi”, 1210], [“Romania”, “Bucharest”, 21],  [“United States”, “Washington”, 307]  print (countries[0][2] / countries[2][2]) -> 64’... |

MUTACIÓN

Podemos cambiar el valor de una lista después de haberla creado.

|  |
| --- |
| p = [“H”, “e”, “l”, “l”, “o”]  p[0] = “Y”  print p -> [“Y”, “e”, “l”, “l”, “o”] |