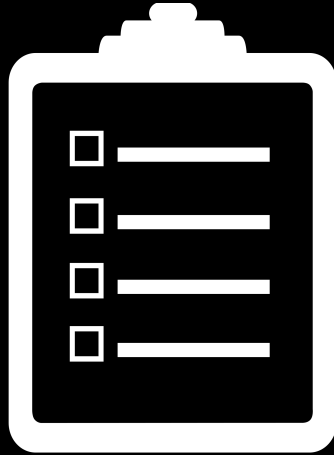


# Mejora paso a paso



Universidad de  
los Andes

# El plan del día



Repaso

Rutina Diaria

Resolver problemas complejos

Demostración: Saltar obstáculos

Repaso

# Entiende cuatros comandos



`move()`

`girar_izquierda()`

`recoger_cono()`

`poner_cono()`

# Definición de un función

```
def nombre_de_funcion():  
    instrucción  
    instrucción  
    ...
```

# Anatomía de un programa

```
1.  """
2.  Un comentario multilínea
3.  Aquí hay otra línea :)
4.  """
5.  def main():
6.      moverse()
7.      # Esto es un comentario de una sola línea
8.      girar_derecha()
9.      moverse()
10.     poner_cono()  # Un otro comentario
11.
12.  def girar_derecha():
13.      girar_izquierda()
14.      girar_izquierda()
15.      girar_izquierda()
16.
```

# Anatomía de un programa

```
1. """
2. Un comentario multilínea
3. Aquí hay otra línea :)
4. """
5. def main():
6.     moverse()
7.     # Esto es un comentario de una sola línea
8.     girar_derecha()
9.     moverse()
10.    poner_cono() # Un otro comentario
11.
12. def girar_derecha():
13.     girar_izquierda()
14.     girar_izquierda()
15.     girar_izquierda()
16.
```

Estos son los  
comentarios. Python va a  
*ignorar* estas  
indicaciones

# Anatomía de un programa

```
1. """
2. Un comentario multilínea
3. Aquí hay otra línea :)
4. """
5. def main():
6.     moverse()
7.     # Esto es un comentario de una sola línea
8.     girar_derecha()
9.     moverse()
10.    poner_cono() # Un otro comentario
11.
12. def girar_derecha():
13.     girar_izquierda()
14.     girar_izquierda()
15.     girar_izquierda()
16.
```

Estos son los comentarios. Python va a *ignorar* estas indicaciones

Los comentarios no son para el programa, sino para humanos!



Los programas son para humanos!

# Un punto sobre los comentarios

# Un punto sobre los comentarios



# El código de Bitcoin

```
bool CBloomFilter::IsRelevantAndUpdate(const CTransaction& tx)
{
    bool fFound = false;
    // Match if the filter contains the hash of tx
    // for finding tx when they appear in a block
    if (vData.empty()) // zero-size = "match-all" filter
        return true;
    const uint256& hash = tx.GetHash();
    if (contains(hash))
        fFound = true;

    for (unsigned int i = 0; i < tx.vout.size(); i++)
    {
        const CTxOut& txout = tx.vout[i];
        // Match if the filter contains any arbitrary script data element in any scriptPubKey in tx
        // If this matches, also add the specific output that was matched.
        // This means clients don't have to update the filter themselves when a new relevant tx
        // is discovered in order to find spending transactions, which avoids round-tripping and race conditions.
        CScript::const_iterator pc = txout.scriptPubKey.begin();
        std::vector<unsigned char> data;
        while (pc < txout.scriptPubKey.end())
        {
            speedtuple speed;
```

# El código de Bitcoin

```
CRollingBloomFilter::CRollingBloomFilter(const unsigned int nElements, const double fpRate)
{
    double logFpRate = log(fpRate);
    /* The optimal number of hash functions is log(fpRate) / log(0.5), but
     * restrict it to the range 1-50. */
    nHashFuncs = std::max(1, std::min((int)round(logFpRate / log(0.5)), 50));
    /* In this rolling bloom filter, we'll store between 2 and 3 generations of nElements / 2 entries.
     nEntriesPerGeneration = (nElements + 1) / 2;
    uint32_t nMaxElements = nEntriesPerGeneration * 3;
    /* The maximum fpRate = pow(1.0 - exp(-nHashFuncs * nMaxElements / nFilterBits), nHashFuncs)
     * => pow(fpRate, 1.0 / nHashFuncs) = 1.0 - exp(-nHashFuncs * nMaxElements / nFilterBits)
     * => 1.0 - pow(fpRate, 1.0 / nHashFuncs) = exp(-nHashFuncs * nMaxElements / nFilterBits)
     * => log(1.0 - pow(fpRate, 1.0 / nHashFuncs)) = -nHashFuncs * nMaxElements / nFilterBits
     * => nFilterBits = -nHashFuncs * nMaxElements / log(1.0 - pow(fpRate, 1.0 / nHashFuncs))
     * => nFilterBits = -nHashFuncs * nMaxElements / log(1.0 - exp(logFpRate / nHashFuncs))
     */
    uint32_t nFilterBits = (uint32_t)ceil(-1.0 * nHashFuncs * nMaxElements / log(1.0 - exp(logFpRate /
data.clear());
    /* For each data element we need to store 2 bits. If both bits are 0, the
     * bit is treated as unset. If the bits are (01), (10), or (11), the bit is
     * treated as set in generation 1, 2, or 3 respectively.
     * These bits are stored in separate integers: position P corresponds to bit
     * (P & 63) of the integers data[(P >> 6) * 2] and data[(P >> 6) * 2 + 1]. */
    data.resize(((nFilterBits + 63) / 64) << 1);
    reset();
}
```

# El código de Bitcoin

```
int nCoinHeight = prevHeights[txinIndex];

if (txin.nSequence & CTxIn::SEQUENCE_LOCKTIME_TYPE_FLAG) {
    const int64_t nCoinTime{Assert(block.GetAncestor(std::max(nCoinHeight - 1, 0)))->GetMedianTimePast()};
    // NOTE: Subtract 1 to maintain nLockTime semantics
    // BIP 68 relative lock times have the semantics of calculating
    // the first block or time at which the transaction would be
    // valid. When calculating the effective block time or height
    // for the entire transaction, we switch to using the
    // semantics of nLockTime which is the last invalid block
    // time or height. Thus we subtract 1 from the calculated
    // time or height.

    // Time-based relative lock-times are measured from the
    // smallest allowed timestamp of the block containing the
    // txout being spent, which is the median time past of the
    // block prior.
    nMinTime = std::max(nMinTime, nCoinTime + (int64_t)((txin.nSequence & CTxIn::SEQUENCE_LOCKTIME_MASK) << CTxIn::SEQUENCE_LOCKTIME_SHIFT));
}
```

# El bucle `for`

```
for i in range(numero):  
    instrucción  
    instrucción  
    ...
```

Repite las instrucciones en el cuerpo del ciclo ***numero*** veces.

# El bucle *while*

*while* *condición*:

*instrucción*

*instrucción*

...

Repite las instrucciones del cuerpo hasta que la *condición* ya no sea verdadera.



# Condiciones posibles

<i>Condición</i>	<i>Opuesto</i>	<i>Qué verifica</i>
frente_despejado()	frente_bloqueado()	¿Hay una pared enfrente de Karel?
izquierda_despejada()	izquierda_bloqueada()	¿Hay una pared a la izquierda de Karel?
derecha_despejada()	derecha_bloqueada()	¿Hay una pared a la derecha de Karel?
conos_presentes()	conos_ausentes()	¿Hay conos en esta esquina?
rumbo_norte()	sin_rumbo_norte()	¿Está Karel orientada hacia el norte?
rumbo_este()	sin_rumbo_este()	¿Está Karel orientada hacia el este?
rumbo_sur()	sin_rumbo_sur()	¿Está Karel orientada hacia el sur?
rumbo_oeste()	sin_rumbo_oeste()	¿Está Karel orientada hacia el oeste?

# Instrucciones condicionales

**if** *condición:*

*instrucción*

*instrucción*

...

Para ejecutar una instrucción  
condicional, usa if

# Instrucciones condicionales

**if** *condición:*

*instrucción*

*instrucción*

**else:**

*instrucción*

*instrucción*

También puedes incluir una  
instrucción else:

# Bloques del código

# Bloques del código

```
def nombre_de_funcion():
```

```
    instrucción
```

```
    instrucción
```

```
    ...
```

```
if condición:
```

```
    instrucción
```

```
    instrucción
```

```
    ...
```

```
while condición:
```

```
    instrucción
```

```
    instrucción
```

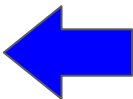
```
    ...
```

## Bloque:

Grupo sentencias consecutivas con el mismo sangrado.

# Bloques del código

```
def nombre_de_funcion():
```

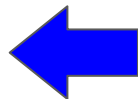


```
    instrucción
```

```
    instrucción
```

```
    ...
```

```
if condición:
```

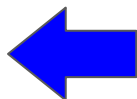


```
    instrucción
```

```
    instrucción
```

```
    ...
```

```
while condición:
```



```
    instrucción
```

```
    instrucción
```

```
    ...
```

## Bloque:

Grupo sentencias consecutivas con el mismo sangrado.

# Ejemplos y contraejemplos

1. `while` frente\_despejado():
2.     `move()`
3.     `poner_cono()`
4.     `move()`

# Ejemplos y contraejemplos

```
1. while frente_despejado():  
2.     | move()   
3.     | poner_cono()  
4.     | move()
```





# Ejemplos y contraejemplos

1. `while` frente\_despejado():
2.     `move()`
3.     `poner_cono()`
4.     `move()`

# Ejemplos y contraejemplos

```
1. while frente_despejado():  
2.     moverse()  
3.     poner_cono()  
4.     moverse()
```



# Ejemplos y contraejemplos

1. `while` frente\_despejado():
2. `move()`
3. `poner_cono()`
4. `move()`



No tienen el mismo sangrado.

# Ejemplos y contraejemplos

1. `while` frente\_despejado():
2. `moverse()`
3. `poner_cono()`
4. `moverse()`

# Ejemplos y contraejemplos

1. `while` frente\_despejado():
2. `move()`
3. `poner_cono()`
4. `move()`



# Ejemplos y contraejemplos

```
1. while frente_despejado():  
2.     moverse()  
3.     poner_cono()  
4.     moverse()
```



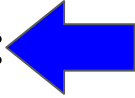
Necesita, **al menos**  
un espacio

# Ejemplos y contraejemplos

```
1. while frente_despejado():  
2.     moverse()  
3.     if cono_presente():  
4.         poner_cono()  
5.         girar_izquierda()  
6.     moverse()
```

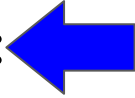
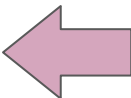
# Ejemplos y contraejemplos

```
1. while frente_despejado():  
2.     moverse()  
3.     if cono_presente():  
4.         poner_cono()  
5.         girar_izquierda()  
6.     moverse()
```





# Ejemplos y contraejemplos

```
1. while frente_despejado():  
2.     | | | | move()   
3.     | | | if cono_presente():   
4.     | | | | poner_cono()  
5.     | | | | girar_izquierda()  
6.     | | | move()
```

Rutina diaria

# Saltar obstáculos

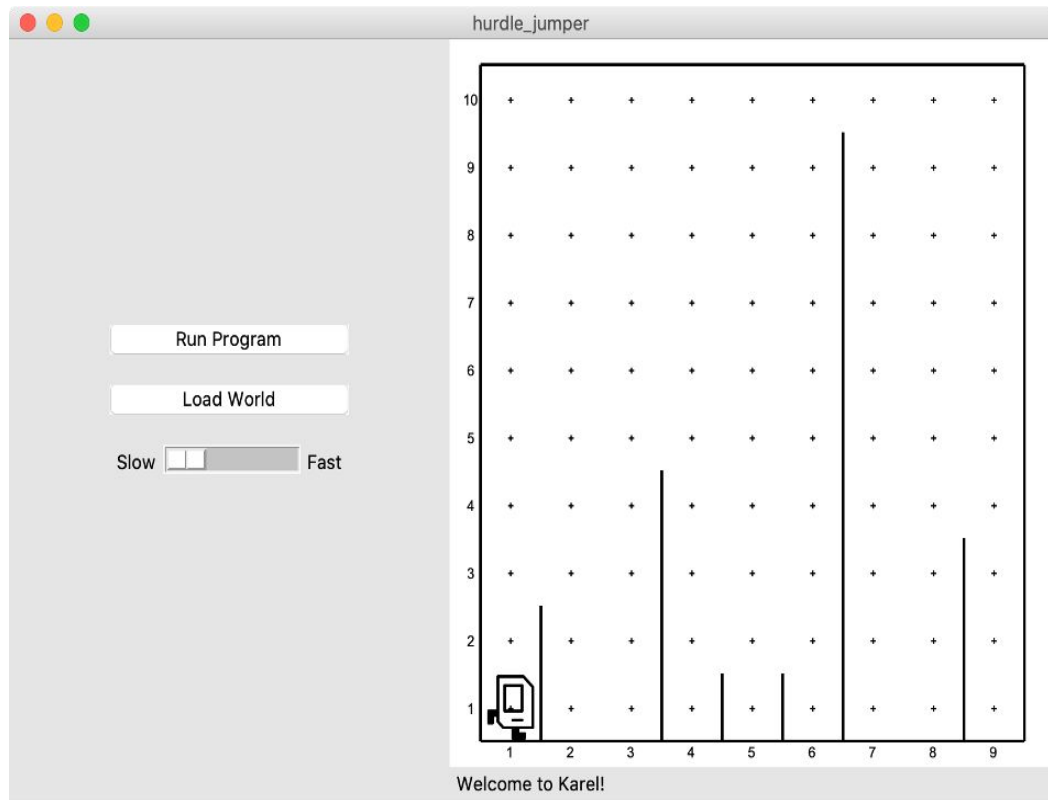
Karel está en los Juegos Olímpicos!

Queremos escribir un programa de Karel que salte obstáculos (vallas).

Karel comienza en (1,1) mirando hacia el este y debe terminar al final de la fila 1 mirando hacia el este.

El mundo tiene 9 columnas.

Hay un número desconocido de "obstáculos" de diferentes alturas que Karel debe ascender y descender para llegar al otro lado.



# Saltar obstáculos

Demostración