Manual Latino

May 12, 2021

Contents

> Have been programing for over four decades. Each programmer needs to be responsible for documenting, testing and debugging what they have created - who else knows it better?
>
> —Prem Sobel

Welcome to the documentation of Latino, the first functional programing language with Spanish syntax.

Author Melvin Guerrero

Translator Melvin Guerrero

Version Latino 1.3.0

CHAPTER 1

Getting Started

About Latino

/Instalar-Latino

/Editores-Textos

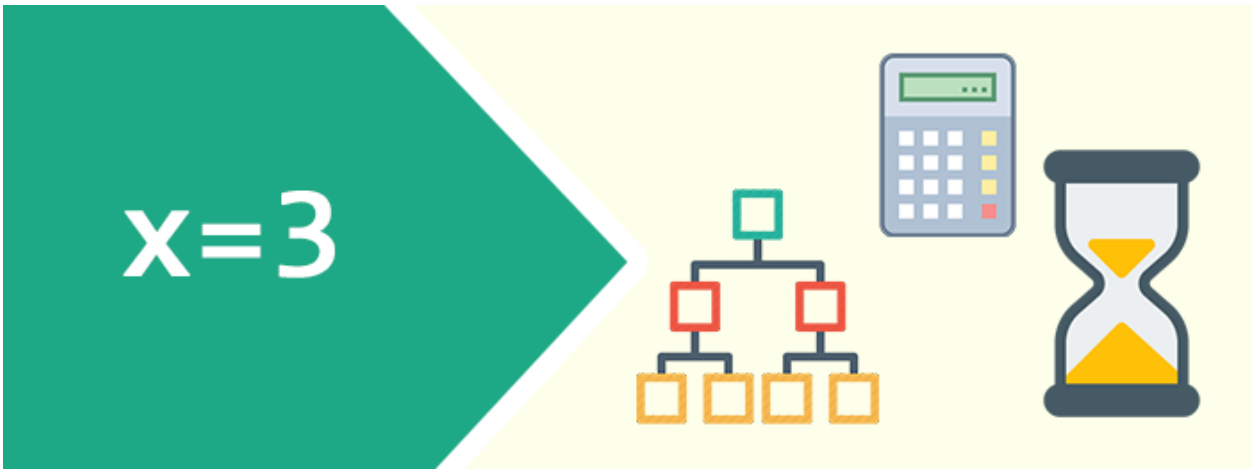
Mi Primer Programa

CHAPTER 2

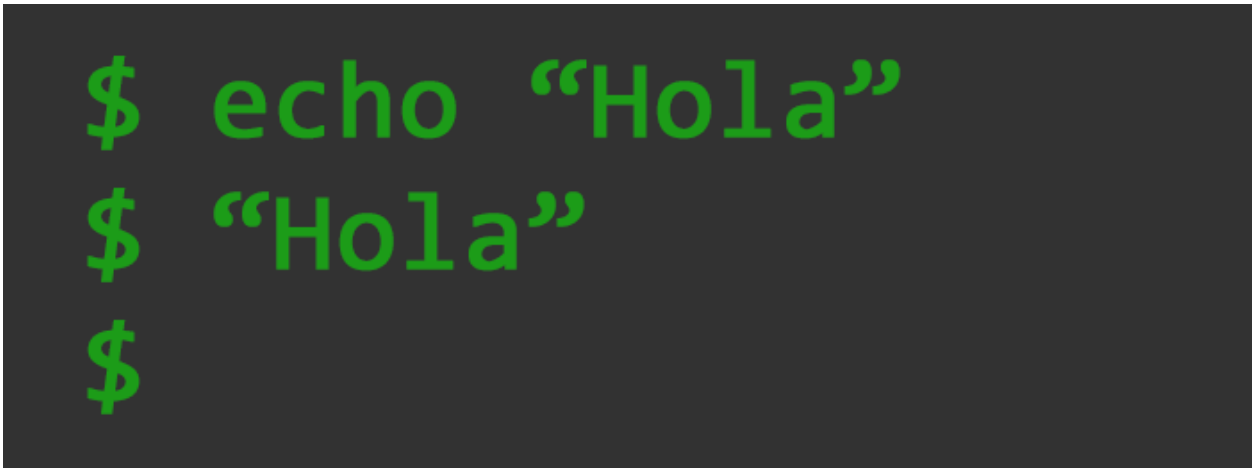
Categories

/Novedades Discover whats new in this version of Latino 1.3.0

Operadores Arithmetic, assignment, conditional, logical, relational operators, and more.



Tipos de Datos Data types is a classification that defines the value associated with a variable or object.



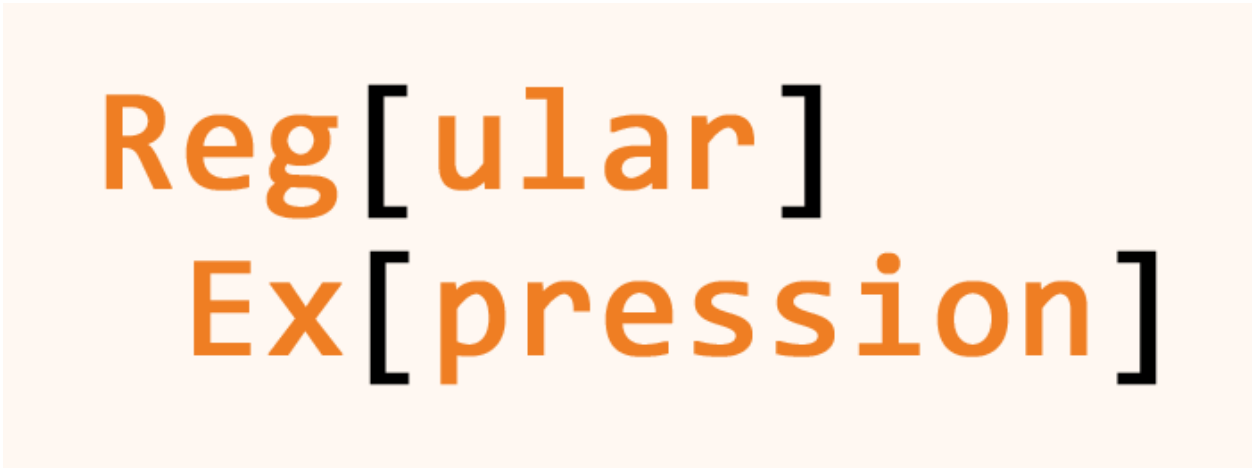
Cadenas (Strings) Strings are used to store and manipulate text in Latino.

Módulos A module is a file that contains a set of functions that you want to include in your application.

RegEx A Regular Expression or RegEx is a sequence of characters that form a search pattern.

/Funciones-Base They are predefined functions that help us perform certain tasks.

Glosario List of libraries, reserved words and definitions in Latino.


Reg[ular]
Ex[pression]

CHAPTER 3

Other Documentations

Latino API (On development)

CHAPTER 4

External links

Youtube channel (Spanish only)

Latino on-line editor

Attention: This documentation is subject to modifications and updates because is still in development. Thank you.

CHAPTER 5

Get Involved

This manual is open to anyone who wants to collaborate.

Please, if you which to help see the collaboration guide.

5.1 Introduction

Code in Spanish!

5.1.1 Who is this intended for?

This documentation is intended for all people that do or don't know programming. It is not important if you have never programmed before or if you are an experienced programmer with desires to program in another language, this programming language is for you.

5.1.2 Learn with examples

A good example is worth more than 1,000 words. This documentation's objective is to show through examples that are easy to understand and remember, helping you to learn and understand this programming language called Latino.

5.1.3 Learn at your pace

It is all up to you. In this documentation, the speed of learning is your choice. If you become overwhelmed, take a break and reread the material for a better understanding. Always assure yourself that you understand all of the examples before advancing to the next module.

5.2 About Latino

5.2.1 What is Latino?

Latino is a fully functional programming language with Spanish syntax, development for this project began in 2015.

This programming language is influenced by Python and Lua.

Latino can be used for:

- Web development (server-side)
- Database connectivity
- Mathematics
- System scripting

5.2.2 Why another programming language?

Latino is born from the need to increase education on computer science in Latino-America; Therefore, students, as well as professionals, could get motivated to enter the world of programming and develop applications with easy-to-learn syntax.

5.2.3 I code in English, why learn Latino?

There is no right answer to this question. If you enjoy experiencing new ways to program and like challenging yourself on a new adventure, then Latino will be a wonderful experience for you!

However, if you are a native Spanish speaker or speak a similar language such as Portuguese or Italian, this could be a great opportunity to learn a super easy-to-learn programming language similar to your own language.

5.2.4 Advantages of this programming language

Latino is influenced by Python and Lua which provide great advantages when using.

1. **Easy Syntax:**
    - *Latino* have a very clean syntax which does not require a semicolon (;) at the end of each line of code which is the case of a programming language such as Java, JavaScript, C/C++, and others.
2. **High-level programming language:**
    - This means that this language looks gramatically similar to how we read and write. However, the Low-level languages are the ones that understand the machine such as Binary code or Assembly language.
3. **Portable:**
    - When we write code in *Latino*, it can be executed on whichever other platfom in a easy and simple way. As opposed to other programming languages like the case of Visual Basic that only works on MS-Windows platforms.

4. **Open source:**

    - Any person can download the source code of *Latino*, modify it, extend the existing libraries, and contribute to its development if they wish.

5.2.5 Who develops Latino?

Latino is a language that has recieved the car and support of Hispanic users around the world and they have contributed to the growth of this project. Users have helped by popularizing and contributing to the source code of the language.

This language was created by Primitivo Roman in 2015 with the version v0.1-alpha until his version v1.0 was launched in 2017.

Currently, the language is in continued development under the direction of Mevlvin Guerrero on the versions v1.1.0, v1.2.0, v1.3.0 from now on.

5.2.6 What are the plans for the future?

Latino continues its development to convert itself to be an essenstial tool in the education of Latin-America and to be good alternative to use in the field of software.

The development of this programming language looks to offer tools that allows the user to create object-oriented applications, mobile applications, videogames, and more!

Also as part of development this language looks to offer educational material of great quality, for the purposes of the use of this language.

For example:

- An accurate manual of great quality
- Books for learning
- Online course
- Video tutorials

5.3 MS-Windows

The installation of *Latino* on Windows is no different from other programs, the process is the exact same. Download the latest version of Latino from the link below which is the lastest release from our github repository.

Download

`Download Latino` Only for **64 bit** systems

---

1. Once the *Latino* installation is downloaded, we proceed to install it by double-clicking it
2. Accept the terms of use
3. De-select the option to create icon on the desktop and select the option **Add PATH**
4. Proceed with the installation

5. Finalize the installation, de-select the run *Latino*

- Once the installation is complete, we test run *Latino* from the CMD to confirm that everything is properly installed. To do this we run CMD on Windows and type the command **Latino** and press Enter. If everything goes well, then CMD will run *Latino* as shown in the image.

Note: If in the process of installing Latino on MS-Windows causes any problems or conflicts, feel free to look for the solutions in the forum here

5.4 Debian - Ubuntu

To install Latino on Debian or Ubuntu , first open the console (Terminal) and execute the following commands:

```
sudo apt-get update
sudo apt-get install git bison flex cmake gcc g++
sudo apt-get install libcurl4-openssl-dev libhiredis-dev libjansson-dev
sudo apt-get install redis-server curl libgtk-3-dev
sudo apt-get install libreadline-dev libpthread-stubs0-dev
```

Note: The code above is **LIBCURL4**, not LIBCUR14

Once this is over, we move on to installing Latino itself in our system

```
cd ~
sudo git clone --recursive https://github.com/lenguaje-latino/Latino
cd latino
sudo git submodule update --init --recursive
sudo cmake .
sudo make
sudo make install
```

... and ready! to run Latino we only need to write in our terminal the **latino** command

Note: If in the process of installing **Latino** on **Debian or Ubuntu** causes any problems or conflicts, feel free to look for the solutions here

5.5 Fedora - CentOS

To install Latino on Fedora or CentOS, first open the console (Terminal) and run the following commands:

Fedora 26

Fedora 25

Fedora 24

```
sudo dnf update
sudo dnf install gcc-c++
sudo dnf install git bison flex cmake kernel-devel
sudo dnf install readline-devel
```

```
sudo dnf update
sudo dnf install gcc-c++
sudo dnf install gtk3-devel
sudo dnf install git bison flex cmake kernel-devel
sudo dnf install hiredis-devel
sudo dnf install readline-devel
```

```
sudo dnf update
sudo dnf install bison flex cmake gcc g++ libjansson-dev libcurl4-openssl-dev␣
↪libhiredis-dev redis-server curl jansson-devel groupinstall "Development Tools"
↪"Development Libraries" groupinstall "RPM Development Tools" redhat-lsb libgtk-3-
↪dev gtk3-devel readline-devel
```

Once this is over, we move on to installing Latino onto the machine

```
cd ~
sudo git clone --recursive https://github.com/lenguaje-latino/Latino
cd latino
sudo git submodule update --init --recursive
sudo cmake .
sudo make
sudo make install
```

... and ready! to run Latino we just write in our terminal the command **latino**

Note: If installing **Latino** in **Fedora or CentOS** causes any problems or conflicts, feel free to look for solutions in the forum 'here'_

5.6 macOS-X

Download

`Download Latino` For Mac with **Intel** processors only, requires **Mac OS X 10.4** or later.

---

1. Once the Latino installer has been downloaded, proceed with the installation
2. Double-click on our Latin-XXX-Darwin installation package.pkg
3. Double-click the icon of the package that appeared on our desktop on Mac
4. Double-click the installation package to start with the installation
5. Follow the instructions in the installation package until finished

---

- Once we finish the installation, we proceed to verify that Latino is properly installed by opening our **Terminal** and in it we will write the command **Latino**

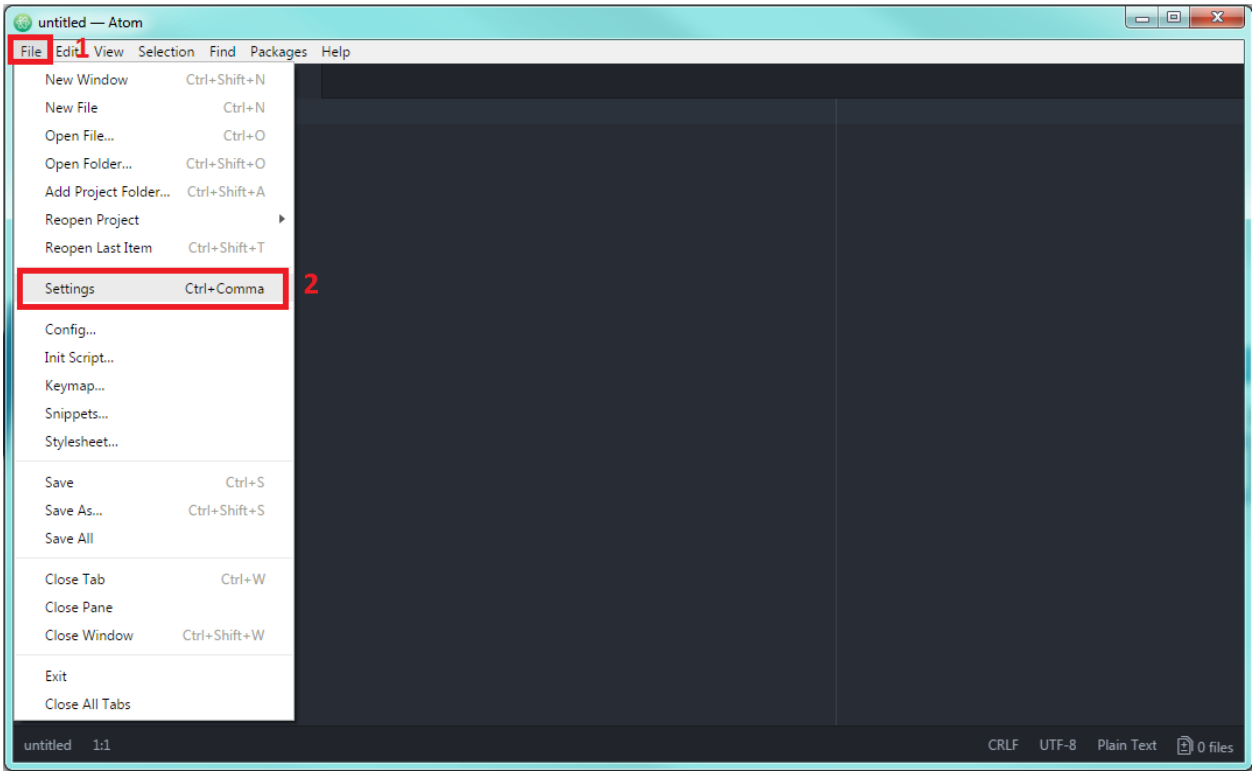
Note: If installing **Latino** on **Mac** causes any problems or conflicts, feel free to look for solutions in the forum 'here'_

5.7 Atom

Sintaxis de Latino en Atom

Para poder usar la sintaxis de Latino en Atom, estos son los pasos a seguir una vez tengamos el programa abierto:

- Clic en Archivo (File) > Configuraciones (Settings)
- Clic en Install > En el buscador escribimos **Latino** y presionamos Enter y por último **Install**
- **y Listo!** Ya podremos programar en Atom con sintaxis de Latino



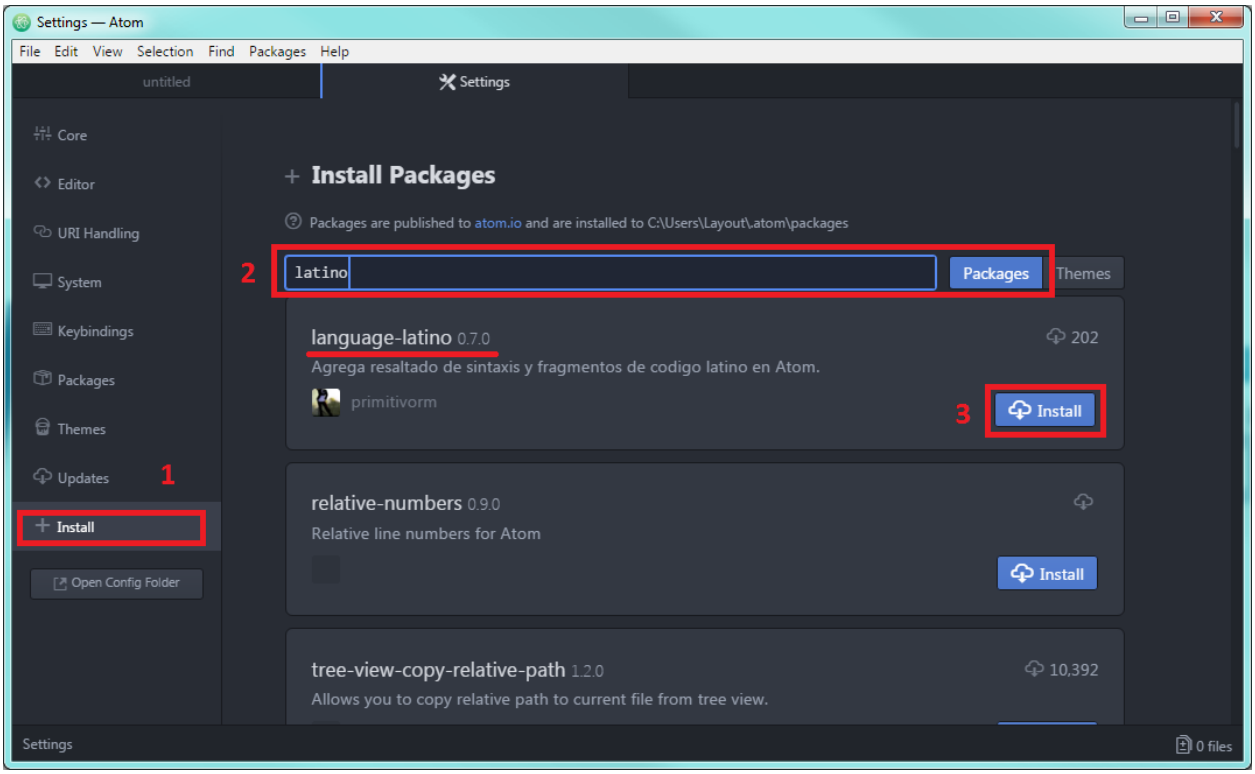
5.8 Notepad++

Descargar

`Descargar Latino-Notepad++`

Sintaxis de Latino en Notepad++

Para poder usar la sintaxis de Latino en Notepad++, estos son los pasos a seguir una vez tengamos el programa abierto:



- Clic en el menú Lenguaje > Definir Idioma
- Clic en el botón **Importar**
- Seleccionamos el archivo **Latino.xml**
- **Reinicie** Notepad++ para que tome la configuración. (Una vez que vuelva a abrir el programa, si la ventana de Definir Idiomas continúa visible lo podemos quitar dando clic en Lenguaje > Definir Idioma)
- Para activar el Plugin de Latino, sólo tendremos que hacer clic en Lenguaje > Latino
- **y Listo!** Ya podremos programar en Notepad++ con sintaxis de Latino

Importante
LLegado a este punto, **reinicie Notepad++**

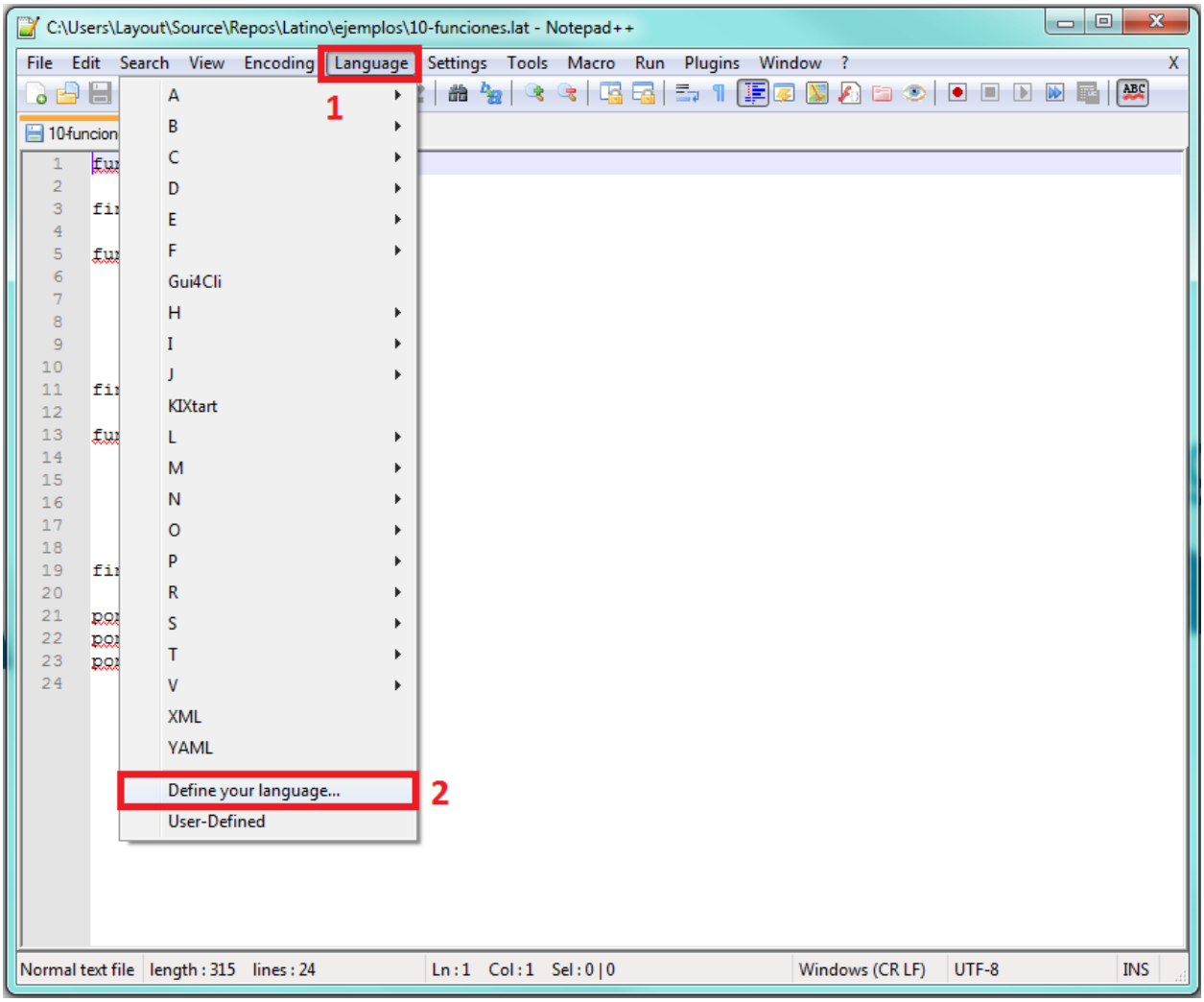
5.9 Sublime Text

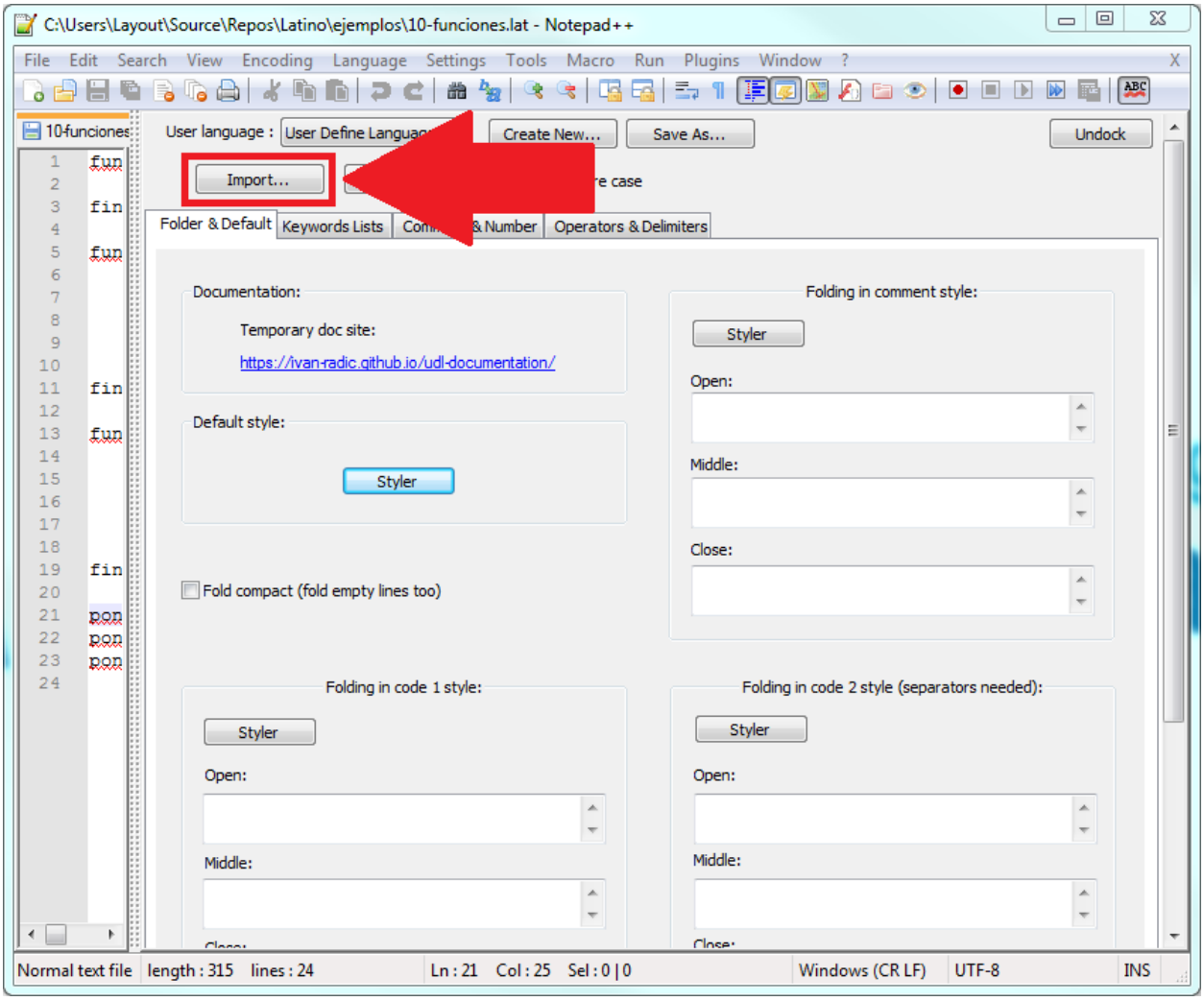
Descargar

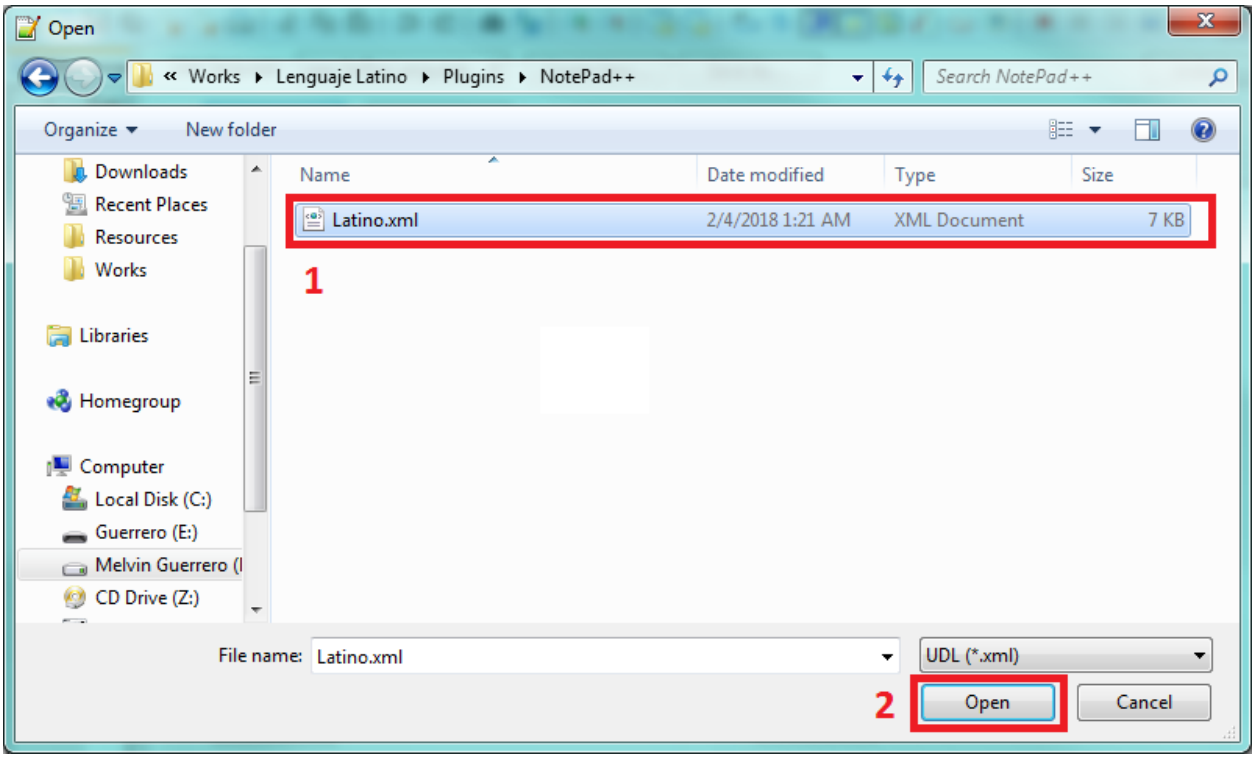
`Descargar Latino-SublimeText`

Sintaxis de Latino en Sublime Text

Para poder usar la sintaxis de Latino en Sublime Text, estos son los pasos a seguir una vez tengamos el programa abierto:


C:\Users\Layout\Source\Repos\Latino\ejemplos\10-funciones.lat - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ? X
1
A
B
C
D
E
F
Gui4Cli
H
I
J
KIXtart
L
M
N
O
P
R
S
T
V
XML
YAML
Define your language...
User-Defined
2
Normal text file length : 315 lines : 24 Ln : 1 Col : 1 Sel : 0 | 0 Windows (CR LF) UTF-8 INS


C:\Users\Layout\Source\Repos\Latino\ejemplos\10-funciones.lat - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
10-funciones
User language :
User Define Languag
Create New...
Save As...
Undock
Import...
Folder & Default
Keywords Lists
& Number
Operators & Delimiters
Documentation:
Temporary doc site:
https://ivan-radic.github.io/udl-documentation/
Default style:
Styler
Fold compact (fold empty lines too)
Folding in comment style:
Styler
Open:
Middle:
Close:
Folding in code 1 style:
Styler
Open:
Middle:
Folding in code 2 style (separators needed):
Styler
Open:
Middle:
Normal text file
length : 315 lines : 24
Ln : 21 Col : 25 Sel : 0 | 0
Windows (CR LF)
UTF-8
INS



- Clic en el menú Preferences > Browse Packages...
- Una vez nos abra el folder, pasaremos a arrastrar y solar (o copiar) los siguientes archivos **LATINO.tmLanguage** y **LATINO.YAML-tmLanguage**
- Reinicie Sublime Text para que tome la configuración.
- **y Listo!** Ya podremos programar en Sublime Text con sintaxis de Latino

5.10 TextMate

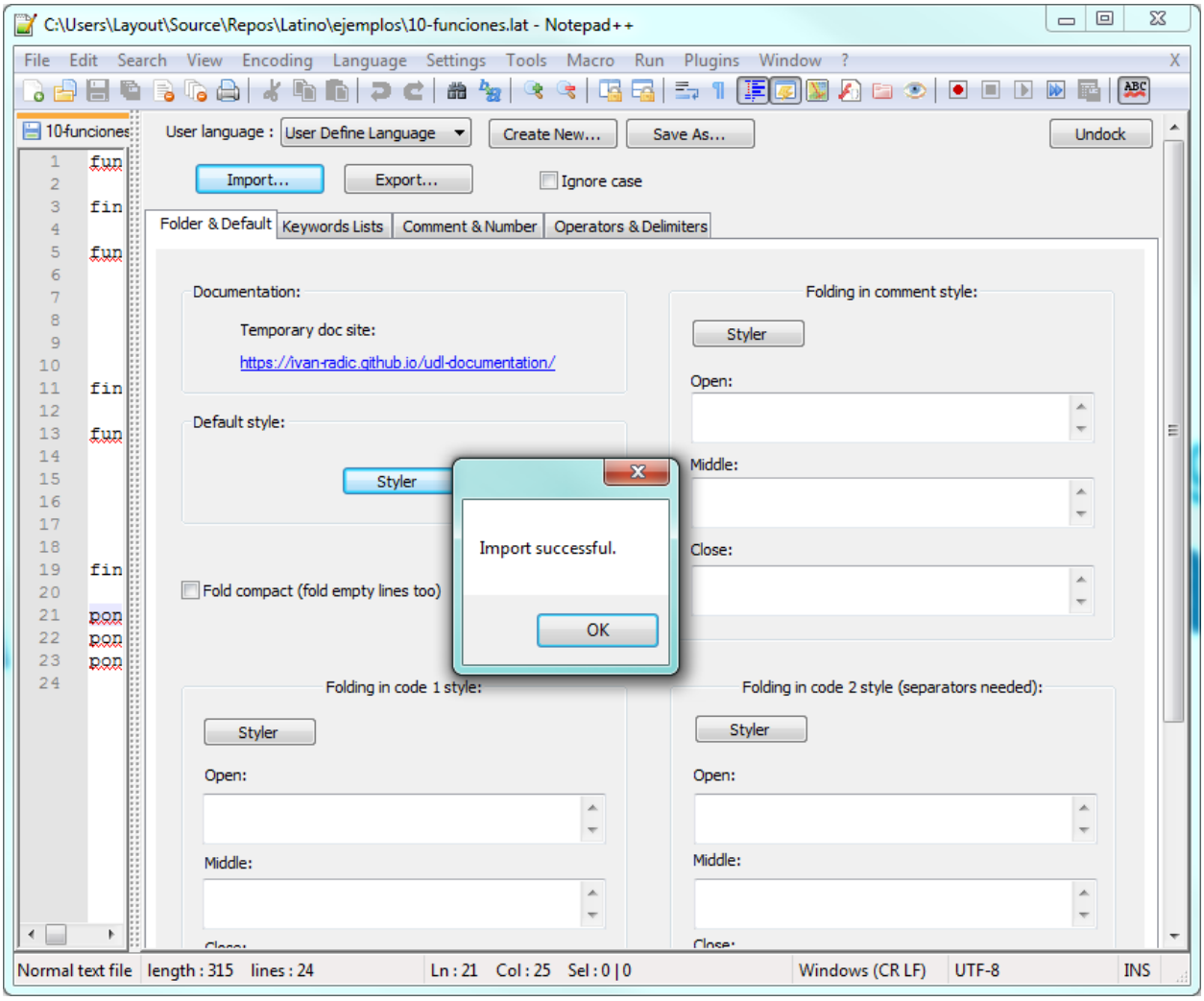
Descargar

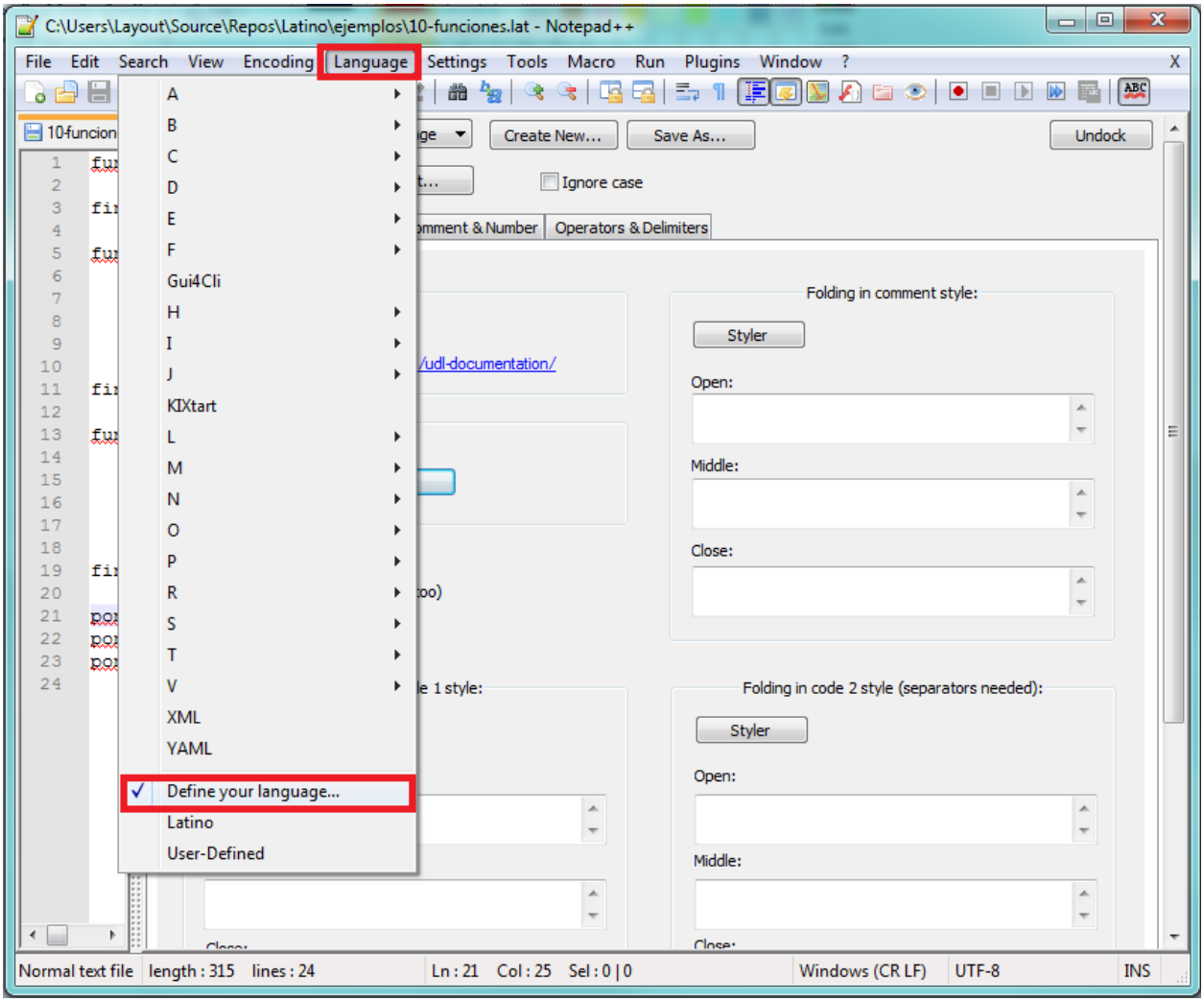
`Descargar Latino-TextMate`

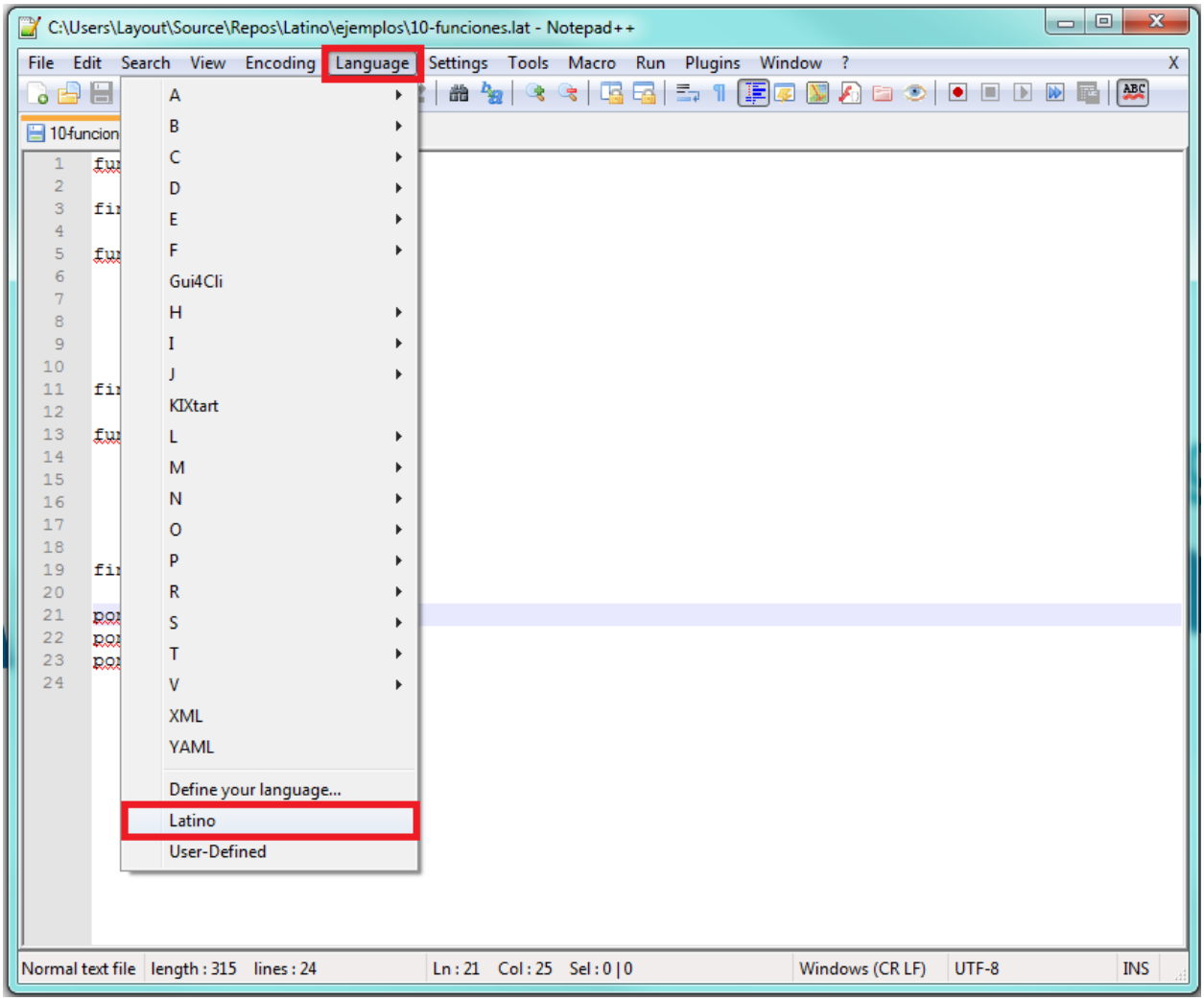
Sintaxis de Latino en Sublime Text

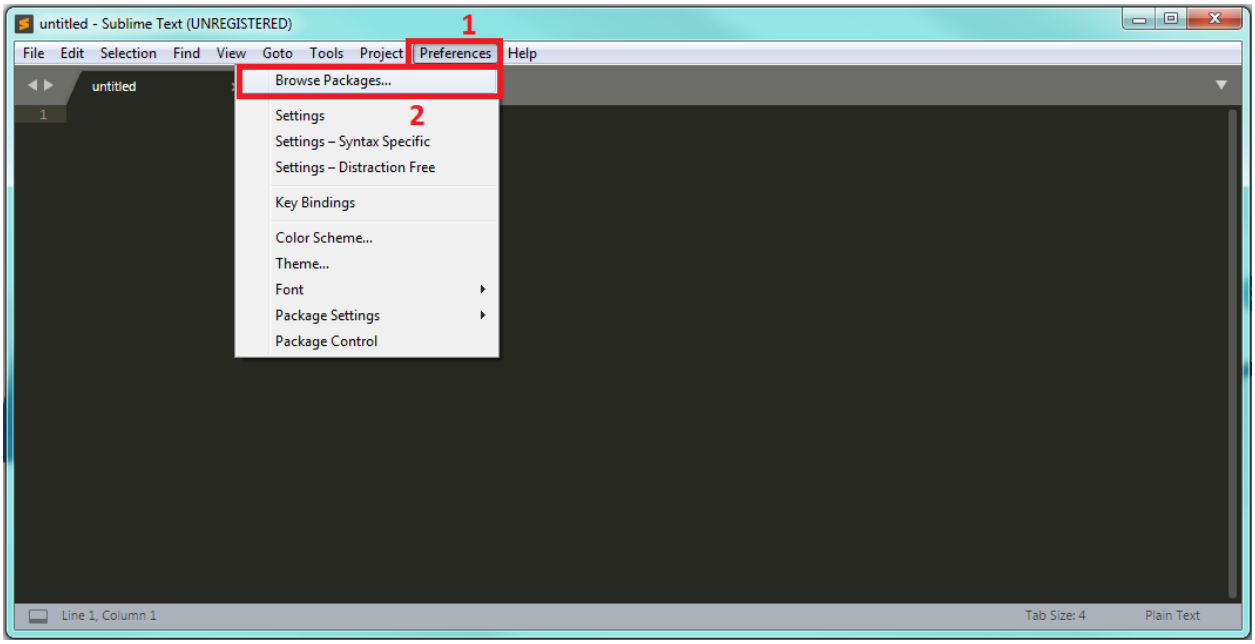
Para poder usar la sintaxis de Latino en TextMate, estos son los pasos a seguir una vez tengamos el programa abierto:

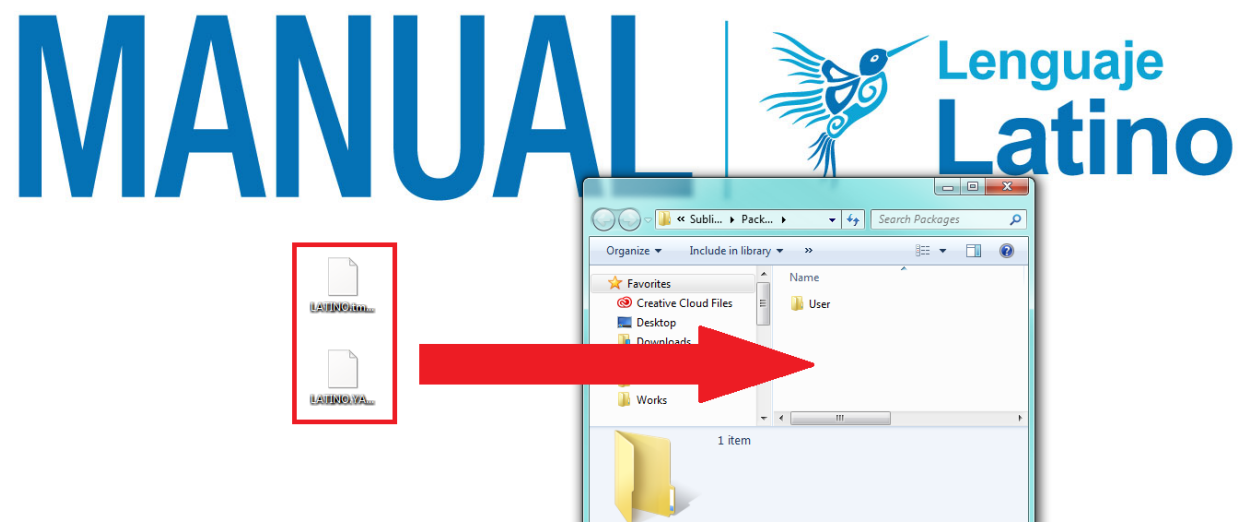
- Descargado el archivo de **latino.tmbundle**
- En TextMate hacemos clic en el menú **Bundles > Edit Bundles...**
- Una vez nos abra la ventana, pasaremos a hacer clic en el menu **File > Open...**
- Pasamos a buscar el archivo **latino.tmbundle**
- Cambiamos el tipo de lenguaje en TextMate
- **y Listo!** Ya podremos programar en TextMate con sintaxis de Latino

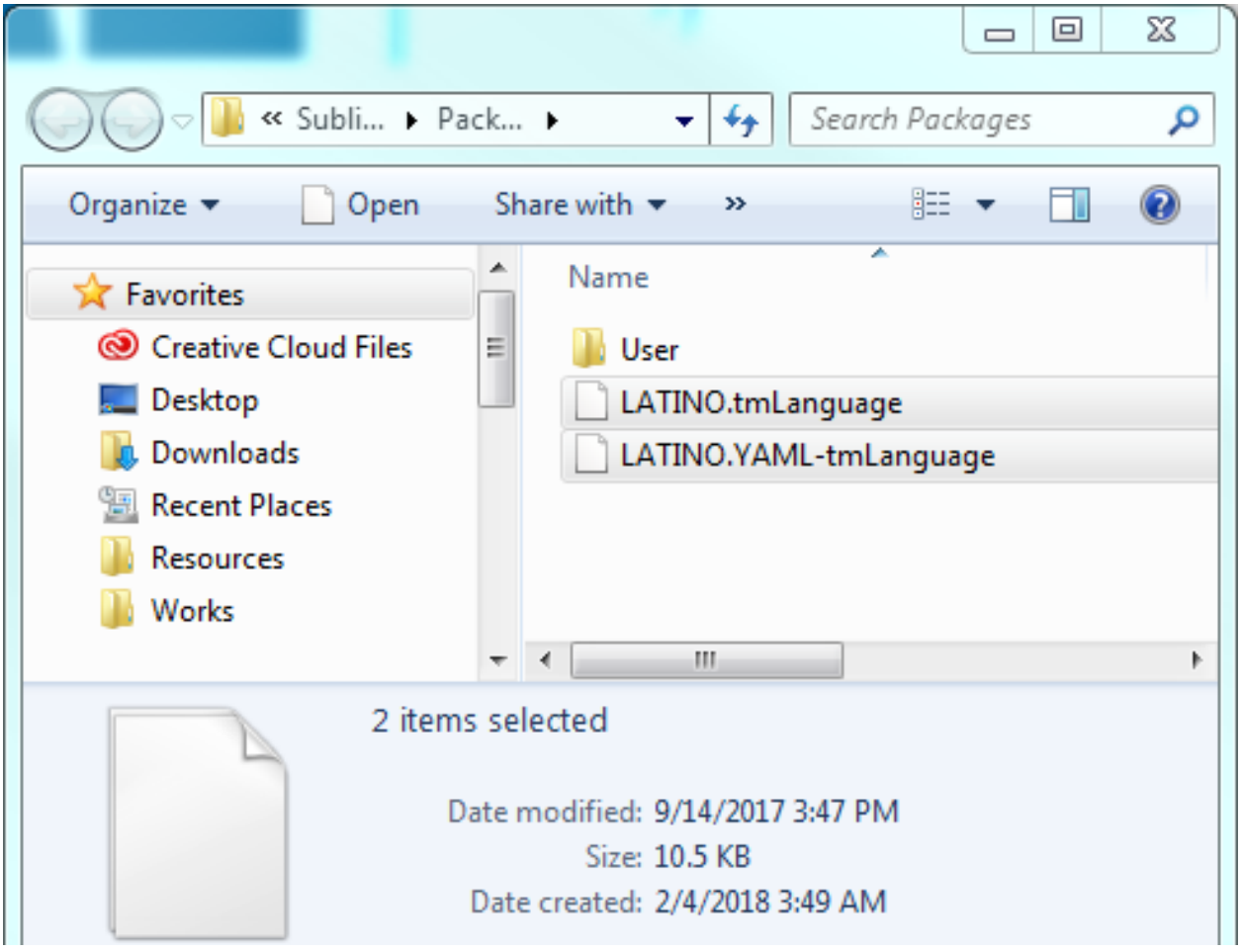

C:\Users\Layout\Source\Repos\Latino\ejemplos\10-funciones.lat - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
10-funciones
User language :
User Define Language
Create New...
Save As...
Undock
Import...
Export...
Ignore case
Folder & Default
Keywords Lists
Comment & Number
Operators & Delimiters
Documentation:
Temporary doc site:
https://ivan-radic.github.io/udl-documentation/
Folding in comment style:
Styler
Open:
Middle:
Close:
Default style:
Styler
Fold compact (fold empty lines too)
Import successful.
OK
Folding in code 1 style:
Styler
Open:
Middle:
Folding in code 2 style (separators needed):
Styler
Open:
Middle:
Normal text file
length : 315 lines : 24
Ln : 21 Col : 25 Sel : 0 | 0
Windows (CR LF)
UTF-8
INS

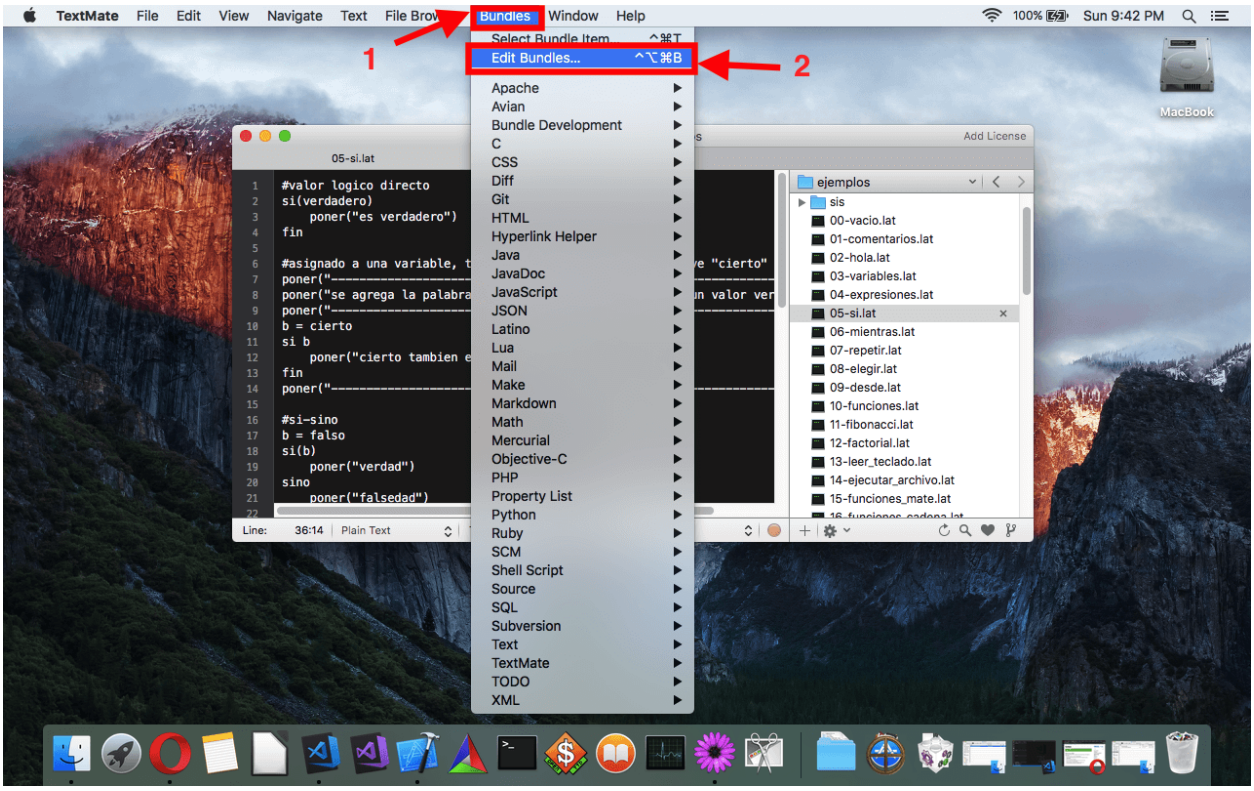

C:\Users\Layout\Source\Repos\Latino\ejemplos\10-funciones.lat - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
A
B
C
D
E
F
Gui4Cli
H
I
J
KIXtart
L
M
N
O
P
R
S
T
V
XML
YAML
Define your language...
Latino
User-Defined
Create New...
Save As...
Undock
Ignore case
Operators & Delimiters
Folding in comment style:
Styler
Open:
Middle:
Close:
Folding in code 2 style (separators needed):
Styler
Open:
Middle:
Normal text file length : 315 lines : 24
Ln : 21 Col : 25 Sel : 0 | 0
Windows (CR LF)
UTF-8
INS

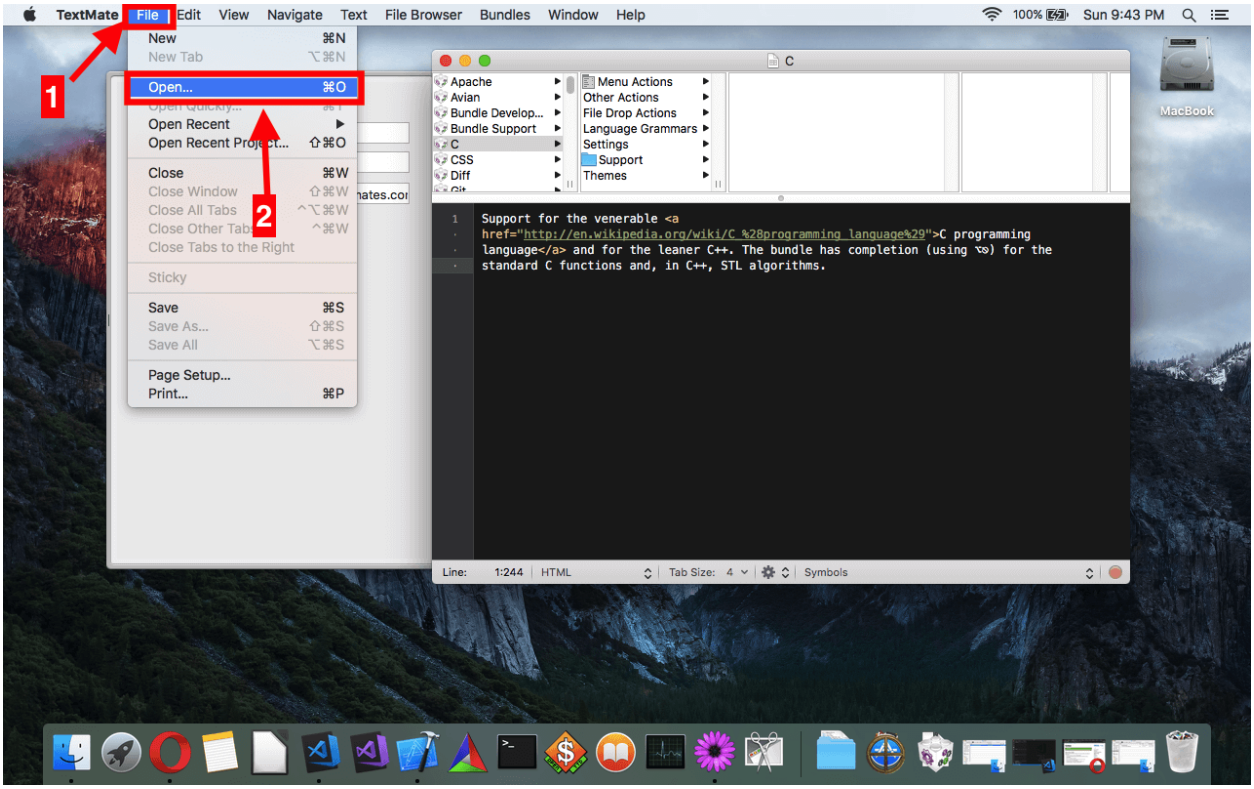

C:\Users\Layout\Source\Repos\Latino\ejemplos\10-funciones.lat - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
A
B
C
D
E
F
Gui4Cli
H
I
J
KIXtart
L
M
N
O
P
R
S
T
V
XML
YAML
Define your language...
Latino
User-Defined
Normal text file length : 315 lines : 24 Ln : 21 Col : 25 Sel : 0 | 0 Windows (CR LF) UTF-8 INS

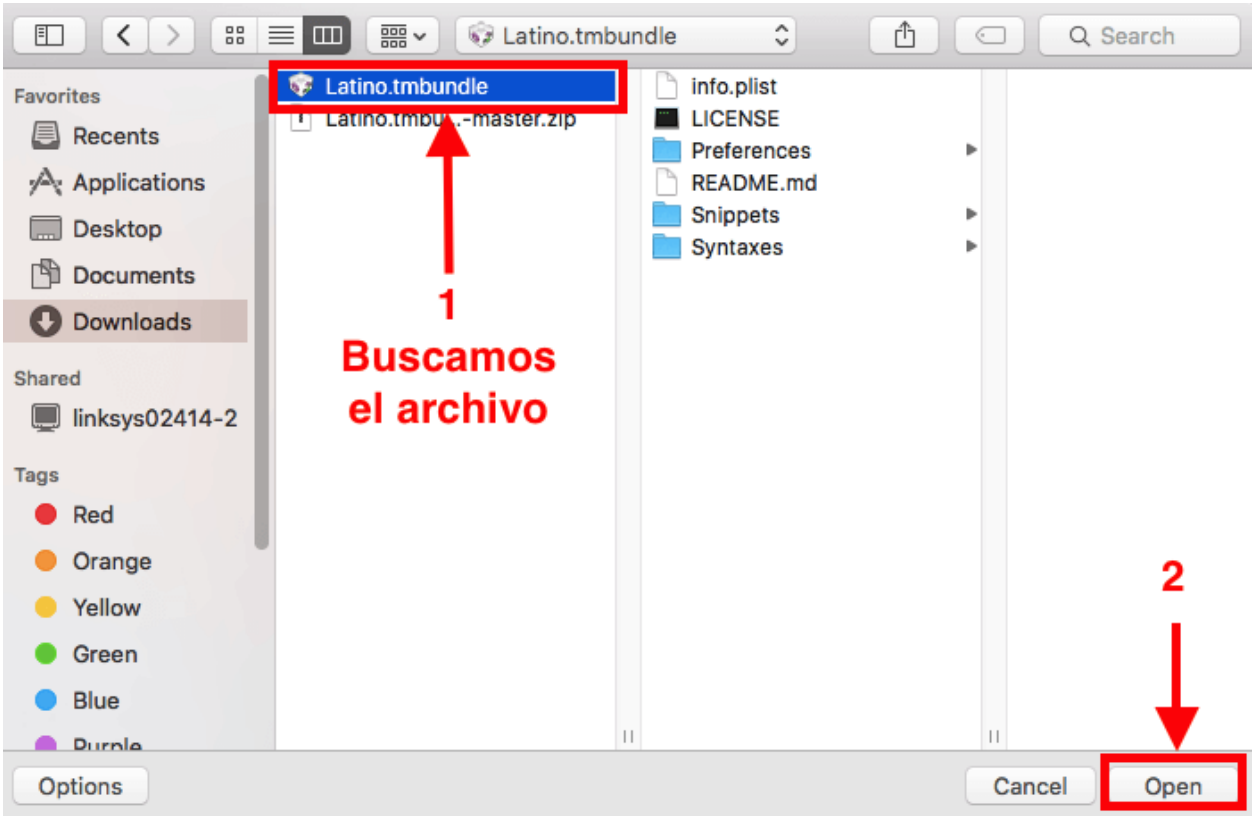

untitled - Sublime Text (UNREGISTERED)
1
File Edit Selection Find View Goto Tools Project Preferences Help
Browse Packages...
Settings
2
Settings – Syntax Specific
Settings – Distraction Free
Key Bindings
Color Scheme...
Theme...
Font
Package Settings
Package Control
Line 1, Column 1
Tab Size: 4
Plain Text

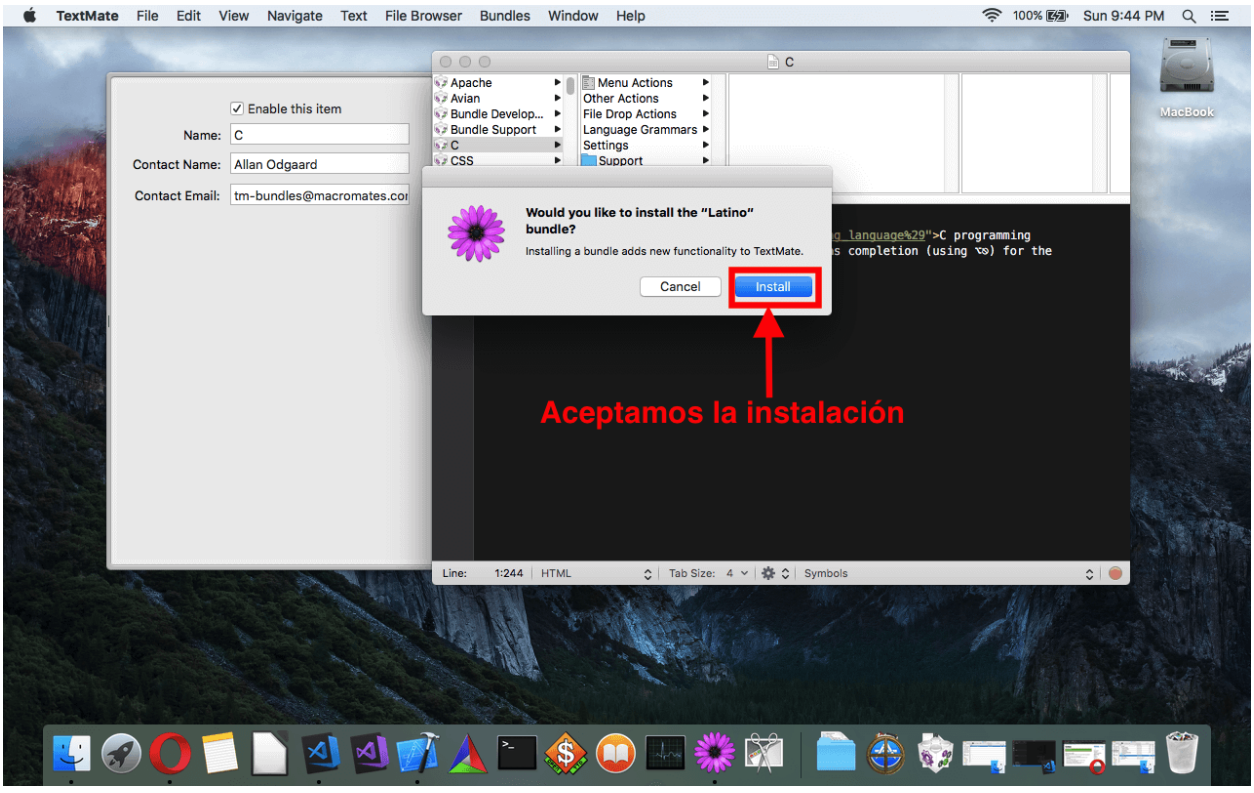

MANUAL
Lenguaje
Latino

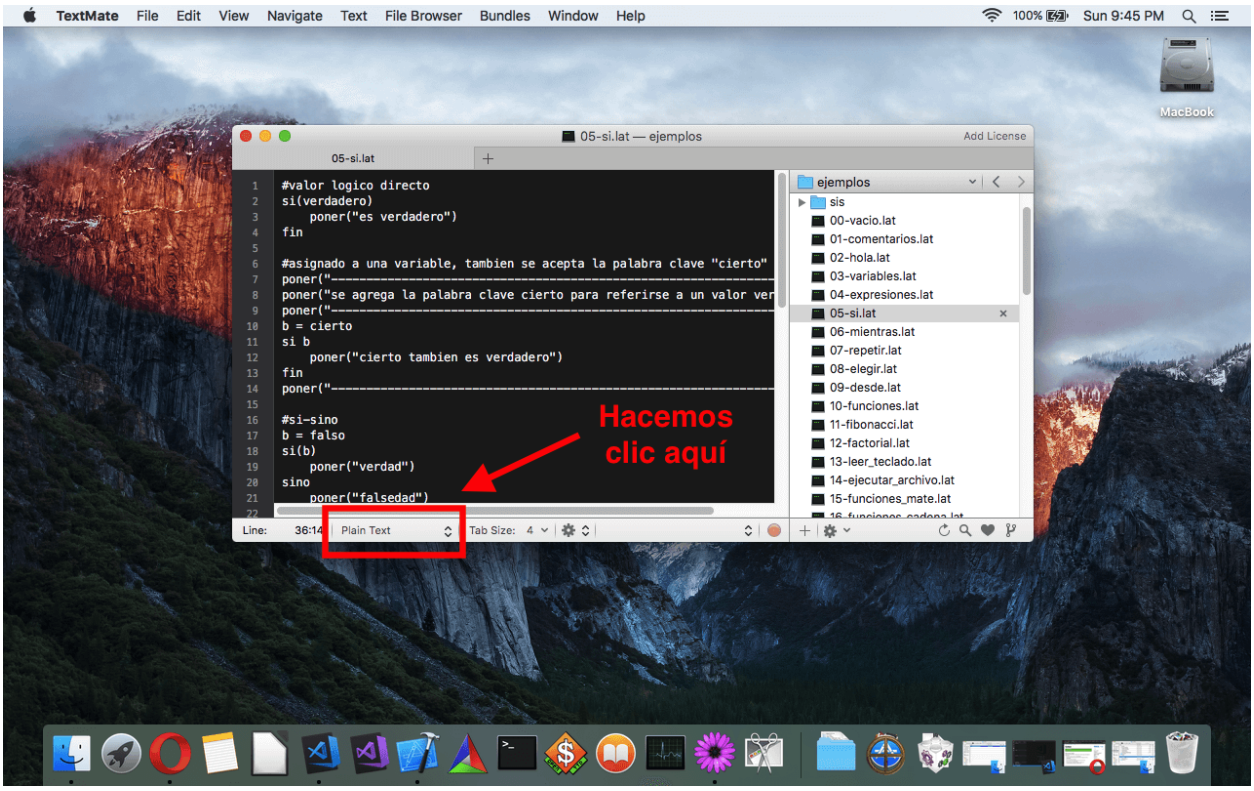

« Subli... ▸ Pack... ▸
Search Packages
Organize
Open
Share with
Favorites
Creative Cloud Files
Desktop
Downloads
Recent Places
Resources
Works
Name
User
LATINO.tmLanguage
LATINO.YAML-tmLanguage
2 items selected
Date modified: 9/14/2017 3:47 PM
Size: 10.5 KB
Date created: 2/4/2018 3:49 AM

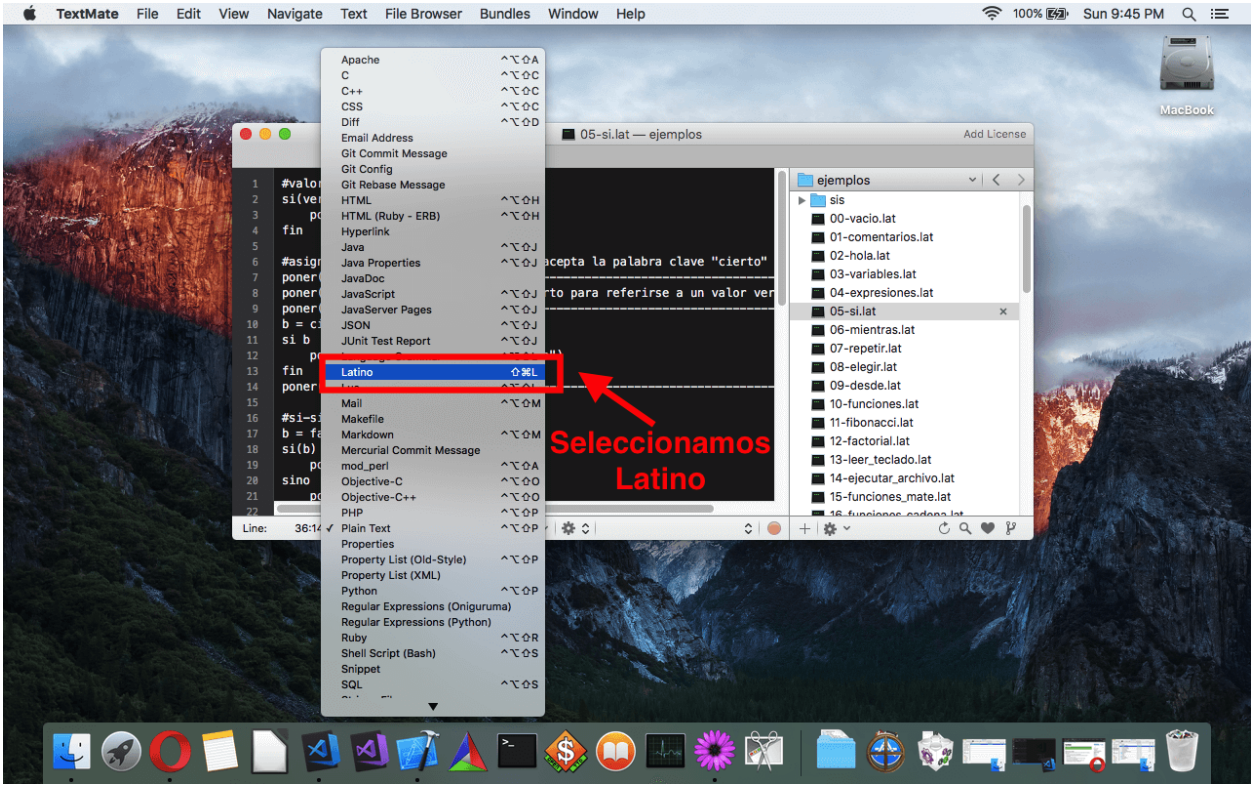

TextMate File Edit View Navigate Text File Browser Bundles Window Help
100% Sun 9:42 PM
1
2
Select Bundle Item ^⌘T
Edit Bundles... ^⌥⌘B
Apache
Avian
Bundle Development
C
CSS
Diff
Git
HTML
Hyperlink Helper
Java
JavaDoc
JavaScript
JSON
Latino
Lua
Mail
Make
Markdown
Math
Mercurial
Objective-C
PHP
Property List
Python
Ruby
SCM
Shell Script
Source
SQL
Subversion
Text
TextMate
TODO
XML
05-si.lat
ejemplos
sis
00-vacio.lat
01-comentarios.lat
02-hola.lat
03-variables.lat
04-expresiones.lat
05-si.lat
06-mientras.lat
07-repetir.lat
08-elegir.lat
09-desde.lat
10-funciones.lat
11-fibonacci.lat
12-factorial.lat
13-leer_teclado.lat
14-ejecutar_archivo.lat
15-funciones_mate.lat
Line: 36:14 Plain Text
MacBook

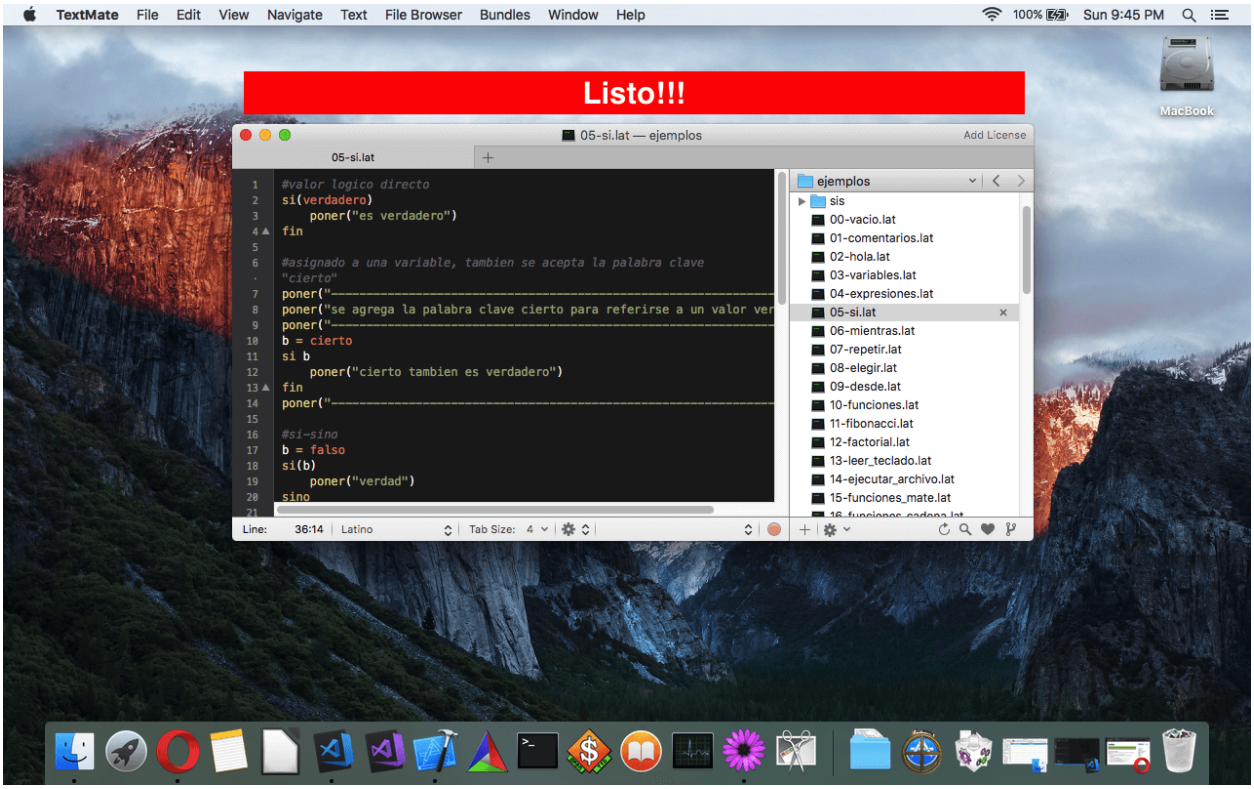

TextMate File Edit View Navigate Text File Browser Bundles Window Help
100% Sun 9:43 PM
1
2
New ⌘N
New Tab ⌥⌘N
Open... ⌘O
Open Recent
Open Recent Project... ⇧⌘O
Close ⌘W
Close Window ⇧⌘W
Close All Tabs ^⌥⌘W
Close Other Tabs ^⌘W
Close Tabs to the Right
Sticky
Save ⌘S
Save As... ⇧⌘S
Save All ⌥⌘S
Page Setup...
Print... ⌘P
C
Apache
Avian
Bundle Develop...
Bundle Support
C
CSS
Diff
Menu Actions
Other Actions
File Drop Actions
Language Grammars
Settings
Support
Themes
Support for the venerable <a
href="http://en.wikipedia.org/wiki/C_%28programming_language%29">C programming
language and for the leaner C++. The bundle has completion (using ⌥⎋) for the
standard C functions and, in C++, STL algorithms.
Line: 1:244 HTML Tab Size: 4 Symbols
MacBook


Latino.tmbundle
Search
Favorites
Recents
Applications
Desktop
Documents
Downloads
Shared
linksys02414-2
Tags
Red
Orange
Yellow
Green
Blue
Latino.tmbundle
info.plist
LICENSE
Preferences
README.md
Snippets
Syntaxes
1
Buscamos
el archivo
2
Options
Cancel
Open


TextMate File Edit View Navigate Text File Browser Bundles Window Help
100% Sun 9:44 PM
Enable this item
Name: C
Contact Name: Allan Odgaard
Contact Email: tm-bundles@macromates.com
Would you like to install the "Latino" bundle?
Installing a bundle adds new functionality to TextMate.
Cancel
Install
Aceptamos la instalación
MacBook


Hacemos
clic aquí


Seleccionamos
Latino



5.11 Vim

Descargar

`Descargar Latino-Vim`

Sintaxis de Latino en Vim

Para poder usar la sintaxis de Latino en Vim, estos son los pasos a seguir:

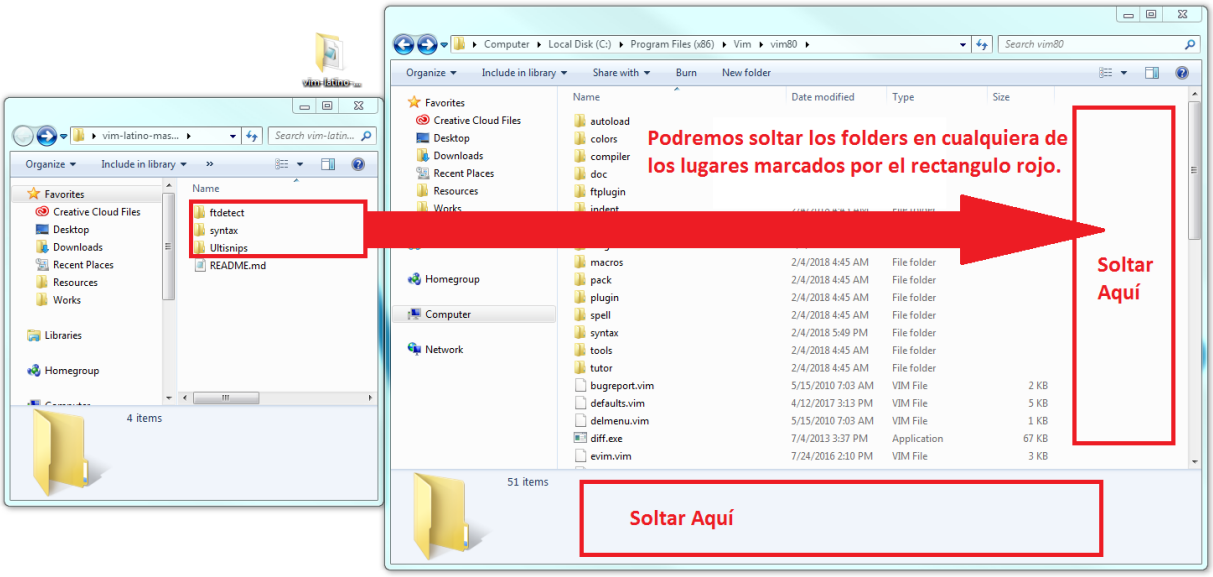
- Con el **programa cerrado**, nos vamos a donde tenemos el programa instalado **C:\Program Files (x86)\Vim\vim80**
- Una vez ahí pasamos a copiar y pegar las carpetas **ftdetect**, **syntax** y **Ultisnips** en la dirección anterior **vim80**
- Aceptamos y confirmamos cuando se nos pregunte si queremos mezclar (**merge**) los archivos.
- **¡Y listo!** Ya podremos programar en Vim con la sintaxis de Latino

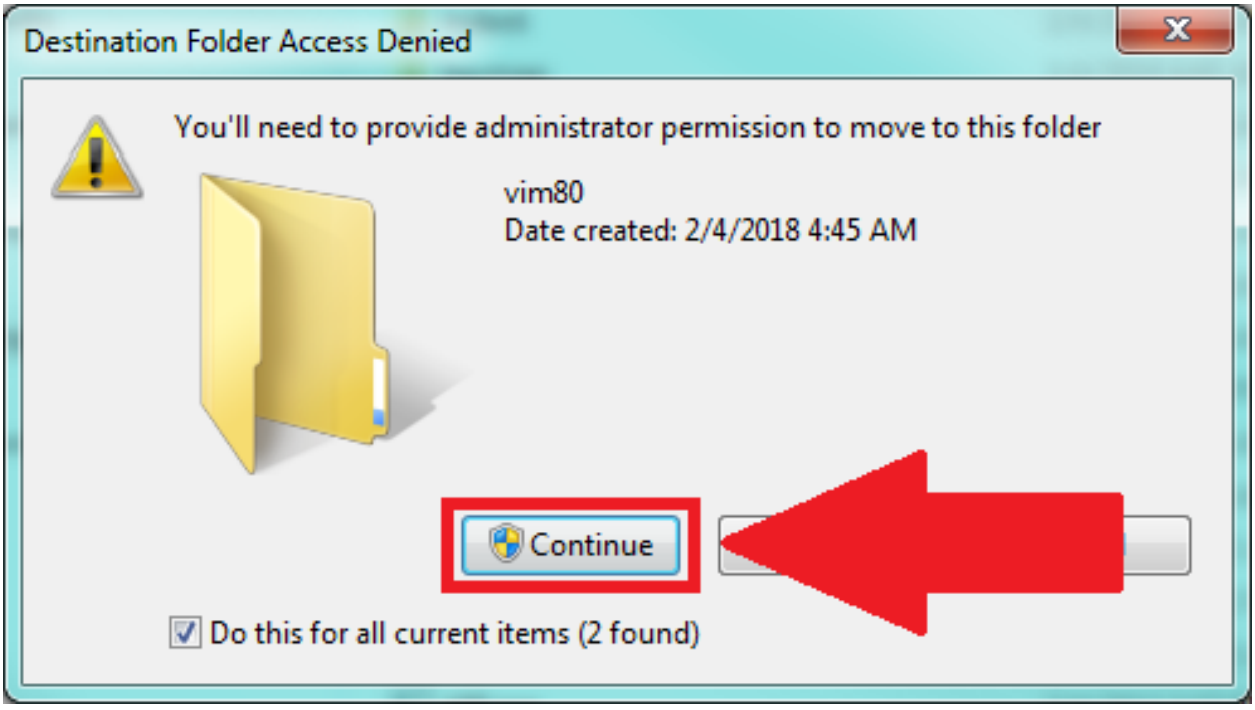
5.12 VS Code

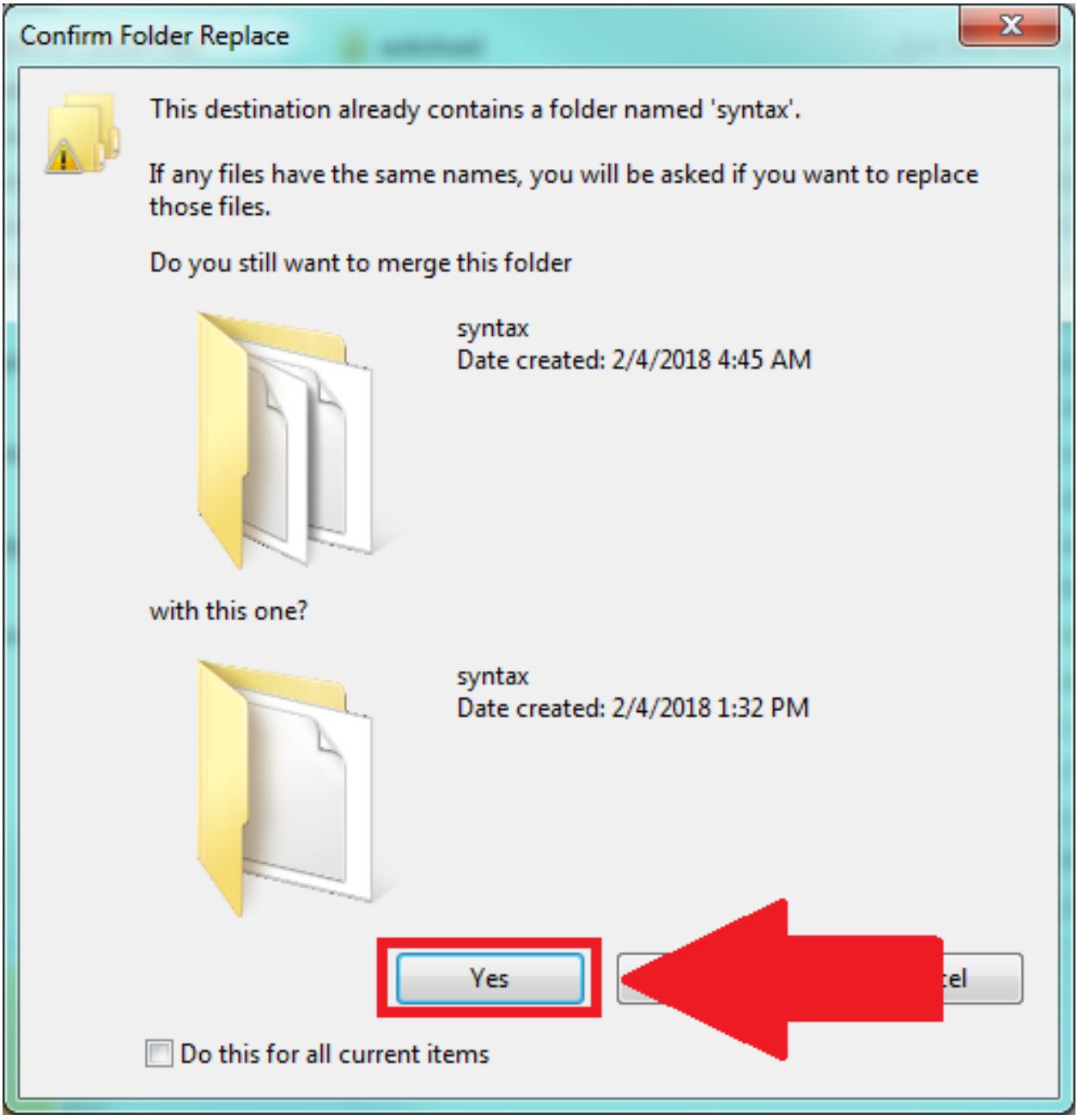
Sintaxis de Latino en VS Code

Para poder usar la sintaxis de Latino en VS Code, estos son los pasos a seguir una vez tengamos el programa abierto:

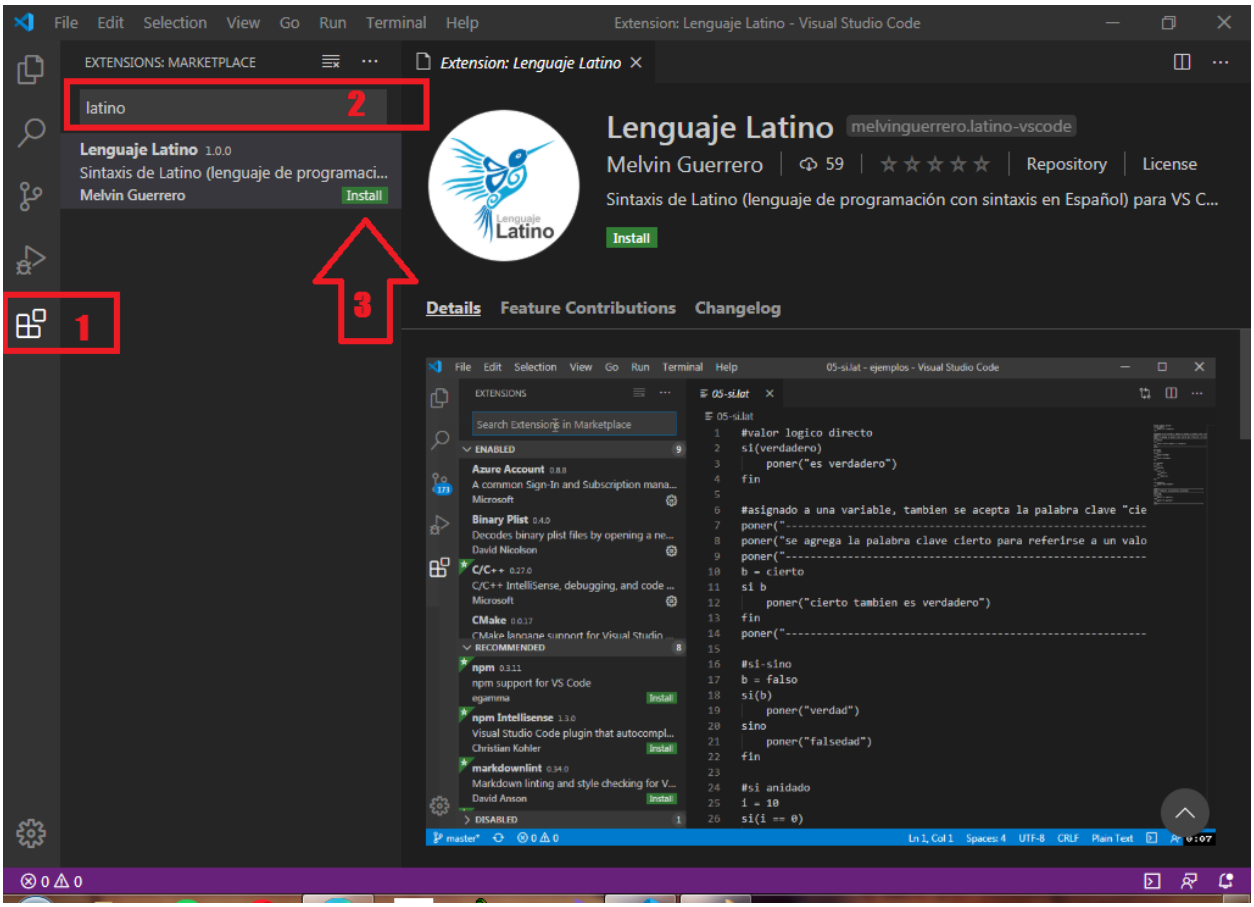
- Clic en Extensiones o precionando su atajo de teclado **Ctrl+Shift+X**
- En el buscador escribimos **Latino** y presionamos Enter y por último **Install**


Podremos soltar los folders en cualquiera de los lugares marcados por el rectangulo rojo.
Soltar Aquí
Soltar Aquí
4 items
51 items


Destination Folder Access Denied
You'll need to provide administrator permission to move to this folder
vim80
Date created: 2/4/2018 4:45 AM
Continue
Do this for all current items (2 found)


Confirm Folder Replace
This destination already contains a folder named 'syntax'.
If any files have the same names, you will be asked if you want to replace those files.
Do you still want to merge this folder
syntax
Date created: 2/4/2018 4:45 AM
with this one?
syntax
Date created: 2/4/2018 1:32 PM
Yes
Do this for all current items

- **y Listo!** Ya podremos programar en VS Code con sintaxis de Latino



5.13 Mi Primer Programa

Esta sección del manual está enfocada en dar al usuario una breve muestra de algunas funciones y operaciones básicas que se pueden hacer en Latino

Para comenzar con cada uno de los ejemplos tendremos que abrir la terminal de nuestro sistema operativo y tener ejecutado Latino.

Note: Para ejecutar Latino en la terminal sólo tendemos que escribir **latino** y precionar la tecla Enter.

Si al escribir el código de Latino se hace uso de un editor de texto, al guardar el documento se requiere guardalos con la extensión **.lat**.
~Ejemplo: **archivo.lat**

Hola Mundo!

Imprimir número

Número par/impar

Intercambiar números

Vocal o Consonante

Hola Mundo en Lenguaje Latino

En este ejemplo vamos a realizar un pequeño programa que al ejecutar mostrara un mensaje que diga **"Hola Mundo, Latino!"**

Para hacer mostrar un mensaje en pantalla usaremos el siguiente comando y presionamos Enter

El resultado será:

```
Hola Mundo, Latino!
```

Imprimir número (digitado por el usuario)

En este ejemplo se mostrara como podemos digitar y almacenar valores a una variable y posteriormente mostrar ese valor en pantalla.

Note: Si está escribiendo el código directamente en la terminal, se puede escribir todo en una sola línea, así:

El resultado será:

```
Digite un número:
24
El número digitado fue: 24
```

Número Par o Impar en Latino

En este ejemplo vamos a crear un programa que nos ayude a identificar cuando un número (digitado por el usuario) es par o impar.

Ejemplo 1:

Note: Si está escribiendo el código directamente en la terminal, se puede escribir todo en una sola línea, así:

El resultado será:

```
Entre un número:
8
El número 8 es par
```

Ejemplo 2:

Esta es otra forma de poder crear el mismo programa pero en menos líneas de códigos:

El resultado será:

```
Entre un número:
-7
El número -7 es impar
```

Intercambiar dos números entre variable en Latino

En este ejemplo haremos un programa que intercambie los valores de dos variables entre si.

Ejemplo 1:

El resultado será:

```
PrimeroNum:5 | SegundoNum: 2
```

Ejemplo 2:

El resultado será:

```
Entre el primer número:
1
Entre el segundo número:
2
Después de intercambiar, la primera variable es de: 2
y la segunda variable es de: 1
```

Ejemplo 3:

El resultado será:

```
Entre el primer número:
10.25
Entre el segundo número:
-12.5
Después de intercambiar, la primera variable es de: -12.5
y la segunda variable es de: 10.25
```

Identificar si el caracter es vocal o no en Latino

En este ejemplo vamos a crear un programa que sea capaz de saber si el valor que insertamos es una vocal o consonante.

El resultado será:

```
Entre un alfabeto:
a
a, es un vocal
```

5.14 Comentarios

Como otros lenguajes de programación, Latino dispone de comentarios. Estos comentarios se pueden usar para crear notas que ayuden a explicar algún código que tengamos escrito o también usarlo para prevenir la ejecución de alguna línea de código al momento de ejecutar el programa.

5.14.1 Comentarios de una línea simple

Los comentarios de una línea simple pueden comenzar con un signo de **#** ó **//**. Cualquier texto o código que este después de estos signos serán ignorados por Latino (No se ejecutaran).

En este ejemplo se hará uso del comentario de una línea antes de cada línea de código:

En este ejemplo se usa un comentario línea simple al final de cada línea para explicar el código:

5.14.2 Comentarios de líneas múltiples

Los comentarios de líneas múltiples comienzan con **/*** y terminan con ***/**. Cualquier texto o código que este dentro de estos signos serán ignorados por Latino (no se ejecutaran).

En este ejemplo se hará uso del comentario de líneas múltiples:

5.14.3 Usando comentarios para prevenir la ejecución de códigos:

El uso de comentarios para prevenir la ejecución de una línea de código puede ser muy útil cuando estamos probando nuestro código. Agregando los signos **#** ó **//** delante de cualquier línea de código, hará que esta única línea se vuelva invisible para el programa al momento de ejecutarlo.

En este ejemplo se hará uso del signo **//** para prevenir la ejecución de la primera linea de codigo:

En este ejemplo se preverá la ejecución de un bloque de código con los comentarios de líneas múltiples:

5.15 Variables

Una variable es un espacio en la memoria, en el cual el programador asigna un valor determinado.

Las variables son representadas por un nombre que es asignado por el programador cuando se escribe el código fuente. Cada variable tiene un único nombre el cual no puede ser cambiado una vez esta variable tenga un valor asignado. Dos o más variables pueden tener el mismo valor o contenido, pero no el mismo nombre.

En este ejemplo **X**, **Y** y **Z** serán las variables:

En este ejemplo, se puede observar la siguiente explicación:

- X es una variable, y esta almacena el valor de 3
- Y es otra variable, y esta almacena el valor de 5
- Z es otra variable, y esta almacena el valor de 8

5.15.1 Declaración (creación) de variables

Las variables de Latino deben ser identificadas por un **nombre único**.

Estos nombres pueden ser cortos (como X o Y) o pueden ser nombres mas descriptivos (como edad, nombre, valorTotal, etc.)

La regla general en Latino para crear nombres de variables son las siguientes:

Las variables SI pueden:

- Empezar con un guión bajo _ o letras **a-z** o **A-Z**.
- Contener caracteres en mayúsculas y minúsculas. (Latino es sensible a las mayúsculas y minúsculas, por lo que los identificadores con nombres similares pero con letras mayúsculas o minúsculas en ellas serán interpretadas como diferentes variables en Latino).

Las variables NO pueden:

- No son validas las letras acentuadas u otros caracteres como la **ñ**.
- Empezar por un número.
- Empezar por un símbolo o alguna *palabra reservada* de Latino.

5.15.2 Declaraciones de múltiples variables

En Latino es posible asignar más de una variable en una sola línea de código. En Latino una declaración múltiple sólo será valida de la siguiente manera:

5.15.3 Tipos de variables

Las variables en Latino pueden contener casi cualquier tipo de datos como cadenas, diccionarios, data, valores numéricos como el número 100 y valores alfanuméricos como un nombre de persona "José Martinez".

En programación, los valores alfanuméricos (o textos) se los llaman **Strings** (por su nombre en inglés).

Los strings se escriben dentro de comillas simples o dobles. Sin embargo, los números se pueden escribir sin estas comillas.

5.15.4 Variables numéricas

En este ejemplo **precio1**, **precio2**, y **total**, serán variables:

Las variables de Latino son contenedores para almacenar variedades de datos:

- En programación, también se usan las variables para guardar valores algebraicos (como es el caso de la variable precio1 y precio2).
- En programación, también se usan las variables para guardar expresiones (como es el caso de total = precio1 + precio2).

5.15.5 Concatenar variables

En Latino, el símbolo **+** no es usado para concatenar datos, como **sí** es el caso de otros lenguajes de programación.

Este símbolo **+** solo es usado para sumar valores numéricos.

Para concatenar datos en Latino se utilizan los puntos dobles **..**

Si se desea concatenar valores alfanuméricos con valores u operaciones numéricas, se recomienda que se declaren en variables separadas y se concatenen en una nueva variable para ser escritas:

5.15.6 Constantes

A este tipo de variables se les llaman **Constantes** porque una vez el programa arranque, su valor no podrá ser reasignado por otro valor, como sí es el caso de las variables anteriores.

Cuando se declara una variable constante, se le debe asignar un valor al momento de declararla, ya que no pueden estar vacias ni se les puede cambiar ni asignar un valor después de que el programa arranque.

Las variables constantes se deben declarar escribiendo todo su nombre en mayúsculas.

5.16 Operadores

En Latino como en otros lenguajes de programación, tiene varios operadores para realizar tareas que incluyen operaciones aritméticas, condicionales y lógicos.

La precedencia de operadores en este lenguaje de programación es la misma que otros lenguajes como C, Lua y Python.

Latino tiene una amplia gama de operadores para realizar diversas operaciones. Para una mejor comprensión de los operadores, estos operadores se pueden clasificar como:

- Operadores aritméticos
- Operadores de incremento y decremento
- Operadores de asignación
- Operadores relacionales
- Operadores lógicos
- Operadores condicionales
- Otros operadores

Note: En Latino estos operadores sólo se pueden usar con variables con **valores numéricos**, no alfanuméricos.

5.16.1 Operadores aritméticos

Los operadores aritméticos realizan operaciones matemáticas básicas tales como suma, resta, multiplicación y división en valores numéricos (constantes y variables).

Operador	Descripción
+	Suma
-	Resta
*	Multiplicación
/	División
%	Modulo (reminente de división)
^	Potencia

Puedes aprender más sobre estos operadores aritméticos en el capítulo de *Aritmética*

5.16.2 Operadores de incremento y decremento

En latino se encuentran dos operadores que incrementan y decrementan el valor numérico de un operando (constante o variable) por 1(uno).

Operador	Descripción
+ +	Incrementa el valor en 1 (valido sólo en post).
- -	Decrementa el valor en 1 (valido sólo en post).

Puedes aprender más sobre estos operadores en el capítulo de *Aritmética*

5.16.3 Operadores de asignación

Los operadores de asignación se usan para asignar un valor a una variable. El operador de asignación más común es **=** (signo de igual).

Operador	Ejemplo	Igual a
=	x = y	x = y
+=	x += y	x = x + y
-=	x -= y	x = x - y
*=	x *= y	x = x * y
/=	x /= y	x = x / y
%=	x %= y	x = x % y

El operador de asignación **+=** agrega un valor a una variable.

Puedes aprender más sobre estos operadores de asignación en el capítulo de *Asignación*

5.16.4 Operadores relacionales

En programación, un operador relacional verifica la relación entre dos operandos. Si la relación es verdadera, devuelve el valor verdadero; si la relación es falsa, devuelve el valor falso.

Los operadores relacionales se utilizan en la toma de decisiones y en los bucles.

Operador	Descripción
==	Igual que
!=	No igual que
>	Mayor que
<	Menor que
>=	Mayor o igual que
<=	Menor o igual que
~=	RegEx (Expresión Regular)

Puedes aprender más sobre estos operadores relacionales en el capítulo de *Relacionales*

5.16.5 Operadores lógicos

Los operadores lógicos se usan para determinar la lógica entre variables o valores y estos devuelven **Verdadero** o **Falso**, dependiendo si la expresión es verdadera o falsa.

Los operadores lógicos se utilizan comúnmente en la toma de decisiones en programación.

Operador	Descripción
&&	**Y lógico.** Sólo será verdadero si todos los operadores son verdaderos.
||	**Ó lógico.** Será verdadero si sólo uno de los dos operadores es verdadero.
!	**NO lógico.** Sólo será verdadero si los operadores son falsos.

Puedes aprender más sobre estos operadores lógicos en el capítulo de *Lógicos*

5.16.6 Operadores condicionales

Un operador condicional es un operador ternario, es decir, funciona en tres operandos.

Sintaxis del operador condicional:

```
(Expresión condicional) ? expresión1 : expresión2
```

El operador condicional funciona de la siguiente manera:

- La primera *expresión condicional* se evalúa primero. Esta expresión se evalúa si es verdadera o si es falsa.
- Si la expresión condicional es verdadera, se evalúa la *expresión1*.
- Si la expresión condicional es falsa, se evalúa la *expresió2*.

5.16.7 Otros operadores

Entre estos operadores podemos encontrar el operador de **concatenación** y el operador de **acceso a miembros**.

Operador de concatenación

El operador de concatenación está representado por **doble punto (..)**.

Este operador se utiliza para concatenar(juntar) expresiones no relacionadas entre sí:

Operador de acceso a miembro

El operador de acceso a miembro está representado por **un solo punto .**

Un claro ejemplo de su uso es cuando trabajamos con *diccionarios* y queremos acceder a sus **propiedades** ó cuando usamos librerías y queremos acceder a sus **métodos**.

5.17 Aritmética

5.17.1 Operadores aritméticos

Los operadores aritméticos realizan operaciones matemáticas básicas tales como suma, resta, multiplicación y división en valores numéricos (constantes y variables).

Operador	Descripción
+	Suma
-	Resta
*	Multiplicación
/	División
%	Modulo (reminente de división)
^	Potencia (Exponencial)

Ejemplos:

Una típica operación aritmética serían con dos números.

Con dos números literales:

o también con variables:

o también con expresiones:

Suma

El operador de **suma** (+):

Resta

El operador de **Resta** (-):

Multiplicación

El operador de **multiplicación** (*):

División

El operador de **división** (/):

Modulo (Remitente)

El operador de **remitente** (%):

Potencia

El operador de **función exponencial** (^):

Note: Para la **potencia**, podemos conseguir el mismo resultado utilizando la librería de matemáticas **mate.pot(5,2)**

5.17.2 Operadores de incremento y decremento

Incrementación

El operador de incremento, se representa por **suma doble** (+ +).

Decrementación

El operador de decremento, se representa por **resta doble** (- -).

5.17.3 Precedencia en los Operadores

En aritmética, todos los operadores (aritméticos, lógicos y relacionales) tienen unas **reglas de precedencia** que se aplican cuando varios operadores actúan juntos, y Latino hace uso de estas reglas.

Los operadores aritméticos, por ejemplo, la multiplicación y la división se ejecutan antes que la suma o la resta.

Para alterar estas reglas de precedencia, se pueden usar **paréntesis ( )**.

5.18 Asignación

Operador	Ejemplo	Igual a
=	x = y	x = y
+=	x += y	x = x + y
-=	x -= y	x = x - y
*=	x *= y	x = x * y
/=	x /= y	x = x / y
%=	x %= y	x = x % y

5.18.1 Operador =

El asignador operacional **=** asigna un valor a la variable **x**

5.18.2 Operador +=

El asignador operacional **+=** suma los valores de las variables **x** más **y** y los asigna a la variable **x**

5.18.3 Operador -=

El asignador operacional **-=** resta los valores de las variables **x** menos **y** y los asigna a la variable **x**

5.18.4 Operador *=

El asignador operacional ***=** multiplica los valores de las variables **x** por **y** y los asigna a la variable **x**

5.18.5 Operador /=

El asignador operacional **/=** divide los valores de las variables **x** entre **y** y los asigna a la variable **x**

5.18.6 Operador %=

El asignador operacional **%=** nos devuelve el remitente (modulo) de la divide los valores de las variables **x** entre **y** y los asigna a la variable **x**

5.19 Relacionales

En programación, un operador relacional verifica la relación entre dos operandos. Si la relación es verdadera, devuelve el valor **verdadero**; si la relación es falsa, devuelve el valor **falso**.

Los operadores relacionales se utilizan en la toma de decisiones y en los bucles (por su nombre en inglés).

Operador	Descripción
==	Igual que
!=	No igual que
>	Mayor que
<	Menor que
>=	Mayor o igual que
<=	Menor o igual que
~=	RegEx (Expresión Regular)

Los *operadores relacionales* y los *operadores lógicos* son utilizados para probar si una operación es verdadera o falsa.

Ejemplos:

Para los siguientes ejemplos supondremos que la variable **x** tiene un valor de **5**

Operador	Descripción	Comparativa	Resultado
==	igual a	escribir(x == 8)	falso
		escribir(x == 5)	verdadero
!=	no igual	escribir(x != 8)	verdadero
		escribir(x != 5)	falso
>	mayor que	escribir(x > 8)	falso
<	menor que	escribir(x < 8)	verdadero
>=	mayor ó igual que	escribir(x >= 8)	falso
<=	menor ó igual que	escribir(x <= 8)	verdadero

Operador ~=

Este operador hace uso de las expresiones regulares(regular expression) en Latino.

Una expresión regular (*regular expression* ó *RegEx*, por su nombre en Ingles) es una secuencia de caracteres y estas forman un patrón de búsqueda.

Note: Las **Expresiones Regulares** (RegEx) contienen un artículo dedicado a su uso en Latino el cual se puede encontrar *aquí*.

5.20 Lógicos

Los operadores lógicos se usan para determinar la lógica entre variables o valores y estos devuelven **Verdadero** o **Falso**, dependiendo si la expresión es verdadera o falsa.

Los operadores lógicos se utilizan comúnmente en la toma de decisiones en programación.

Operador	Descripción
&&	**Y lógico:** Sólo será verdadero si todos los operadores son verdaderos.
||	**Ó lógico:** Será verdadero si sólo uno de los dos operadores es verdadero.
!	**NO lógico:** Sólo será verdadero si todos los operadores son falsos.

Ejemplo:

Para los siguientes ejemplos supondremos que la variable **x=6** y la variable **y=3**.

Operador	Descripción	Ejecución	Resultado
&&	y lógico	escribir(x < 10 && y > 1)	Verdadero
||	o lógico	escribir(x == 5 || y == 5)	Falso
!	no lógico	escribir(!(x == y))	Verdadero

5.21 Tipos de Datos

Las computadoras a diferencia de los seres humanos, no reconocen ni saben la diferencia entre "1234" y "abcd", por esta razón en programación se definieron los **tipos de datos**.

Un tipo de dato es una clasificación que define el valor asociado a una variable u objeto. Por lo general estos tipos de datos suelen estar almacenados en una variable, ya que estas pueden almacenar tipos de datos como: Numéricos, alfanuméricos, listas, diccionarios, entre otros.

A continuación se presenta una tabla con ejemplos de algunas clasificaciones para los tipos de datos en programación tanto para Latino y C.

Latino	Tipo de datos en C	Ejemplos
lógico	bool	verdadero ó falso
numérico (decimal)	double	1.69549875
cadena	char*	letras
lista (matriz)	array	agr1, agr2, agr3...
diccionario	struct	"propiedad": "valor"
nulo	void	vacio (no data)

5.21.1 Tipos de datos: Lógico

Los tipos de datos lógicos (o booleans por su nombre en inglés) solo pueden tener dos valores: **verdadero** o **falso**.

5.21.2 Tipos de datos: Numérico

Los tipos de datos numéricos son números asignados a una variable que se pueden escribir con o sin **punto decimal**.

Note: También se pueden escribir en **notación científica**, ejemplo:

5.21.3 Tipos de datos: Cadena

Los tipos de datos alfanuméricos (o strings por su nombre en inglés) son líneas de textos escritas entre **comillas simples o dobles**. Ejemplo "Bill Gates" 'Steve Jobs'.

5.21.4 Tipos de datos: Lista

Los tipos de datos de listas o matrices se escriben entre **corchetes [ ]** y sus elementos están separados por **comas**.

Las matrices son indexadas desde el número 0 (Cero) en adelante.

Esto significa que el número de índice del primer elemento será [0], y el segundo será [1], y así sucesivamente.

5.21.5 Tipos de datos: Diccionario

Los tipos de datos de diccionarios u objetos se escriben entre **llaves { }** y sus propiedades se escriben **"propiedad" : "Valor"** y cada propiedad con su valor están separadas por **comas**.

5.21.6 Nulo

En Latino los tipos de datos **Nulos** son nada. Se supone que es algo que no existe.

En Latino una variable con valor nulo **no es igual** a una variable con **valor indefinido**.

Las variables con valor indefinido son imposibles de declarar(crear) en Latino ya que las variables requieren ser asignadas a un valor al momento de estas ser declaradas, de lo contrario Latino solo las omitirá.

De todas maneras si se pueden llegar a crear variables con valor **vacío** de la siguiente manera:

Diferencia entre un valor vacío y un nulo

Un valor vacío y un valor nulo en primera instancia puede que aparenten ser lo mismo ya que nos permite asignar un valor no definido a una variable, pero internamente son totalmente distintas.

Operador "tipo()"

En Latino podemos usar el operador **tipo()** para saber el tipo de dato que lleva un objeto o una variable.

5.22 Números

En diversos lenguajes de programación existen varias clasificaciones para los números entre ellas están integer, float, double y complex.

Latino trabaja solo con un tipo de número, los números que se escriben con decimal y los que no.

Error: Las siguientes expresiones científicas no son reconocidas por Latino:

Note: Latino dispone de una librería para matemáticas la cual puede ver *aquí*.

5.22.1 Decimales

Todos los números en Latino siempre serán reconocidos como decimales, tengan o no el punto decimal.

Esto lo podemos comprobar utilizando la función **tipo( )** en Latino.

Los números también puede ser expresados como números negativos:

Note: Los números pasados de 15 dígitos son devueltos como expresión científica en Latino:

5.22.2 Números como cadedas(textos)

A diferencia de las cadenas los números no requieren ser escritos entre comillas, pero de serlo estas dejaran de ser números y pasaran a ser interpretadas como cadenas(textos).

Convertir cadenas a números

Latino puede convertir las cadenas numéricas a números con los operadores aritméticos:

En Latino existe una funcion pre-definida llamada **anumero( )** que de igualmanera nos convierte una cadena a número.

5.23 Cadenas (Strings)

Las **cadenas** (*strings* por su nombre en Ingles) son utilizadas para almacenar y manipular textos.

Estas **cadenas** están rodeadas por **comillas simples (')** o **comillas dobles (‘)**.

```
escribir("hola")
escribir('hola')
```

Se pueden usar comillas dentro de una cadena, siempre y cuando estas no coincidan con las comillas que las rodean.

Note: Latino dispone de una librería para el manejo de cadenas, *aquí*.

5.23.1 Concatenar cadenas(textos)

Para concatenar o unir textos en Latino se hace uso de **doble puntos (..)**, que a diferencia de otros lenguajes de programación los cuales usan el signo de **más (+)**.

El **doble punto (..)** no solo es útil para unir textos, también números.

5.23.2 Caracteres especiales

Las cadenas al estar escritas entre comillas, se pudiera producir un error con la siguiente sintaxis:

En el ejemplo anterior, La oración estaría cortada hasta 'Hola mundo,' y Latino pensaría que el nombre **Latino!** es una variable, produciendo así un error de sintaxis.

La solución a este problema sería usar una **barra invertida (\ )**.

La **barra invertida (\ )** convierte los caracteres especiales en textos:

Además de usar la barra invertida para escribir textos, también es usada para indicar funciones.

A continuación se presentan los caracteres disponibles para usar en Latino.

Tabla de caracteres:

Carácter	Descripción
\ ‘	Comillas dobles
\ ’	Comillas simples
\n	Nueva línea
\r	Retorno de carro (Carriage return)
\b	Remover (Backspace)
\t	Tabulación horizontal
\v	Tabulación vertical
\f	Alimentación de formulario (Form feed)
\a	Alerta (Beep)
\0	Carácter nulo
\nnn	Carácter con valor octal nnn

Note: Los caracteres especiales descritos en la tabla superior fueron originalmente diseñados para el uso de máquina de escribir, teletipo, y máquina de fax.

Comillas dobles

Carácter a usar: **\ ‘**

Comillas simples

Carácter a usar: **\ ’**

Nueva línea

Carácter a usar: **\n**

Retorno de carro

Carácter a usar: **\r**

Note: Para saber la diferencia entre **\n** y **\r** ver enlace aquí

Remover

Carácter a usar: **\b**

Tabulación horizontal

Carácter a usar: **\t**

Alerta

Carácter a usar: **\a**

Carácter nulo

Carácter a usar: **\0 *(cero)**

Note: De igual manera **\0** al ser un valor nulo, también puede ser usado en condicionales lógicas:

Ejemplo 1

Ejemplo 2

Carácter con valor octal

Carácter a usar: **\nnn**

5.23.3 Textos de multiples líneas

Previamente vimos que las cadenas pueden ser textos líneales, pero también pueden ser textos de multiples líneas.

Para indicar cuando una cadena será de múltiples líneas, basta con dejar una de las comillas al inicio y otra al final del párrafo.

5.23.4 Textos como listas

Como en muchos otros lenguajes de programación, los textos son *listas (arrays)*.

5.23.5 Convertir números a cadenas

En Latino existe una funcion pre-definida llamada **acadena( )** que nos convierte un número a una cadena(texto).

5.24 Funciones

Una función es un bloque de código que realiza una tarea específica.

Una función se ejecuta cuando esta es invoca (llamada).

Una ventaja que propone crear y usar una función es la división de problemas complejos en pequeños componentes que hacen el programa mas fácil de entender y programar.

Cuando estamos programando y tenemos líneas de códigos que se repiten o necesitamos de ciertos códigos en más de una ocasión, la mejor forma de gestionar nuestro código es, creando funciones.

5.24.1 Sintaxis de función

Las funciones se definen con la palabra clave de **funcion** o la forma corta **fun**, seguido por el **nombre de la función** y terminando con **paréntesis()**.

Ejemplo de sintaxis

```
funcion nombre1()
  #código
fin

fun nombre2()
  #código
fin
```

Los nombres de funciones SI pueden:

- Los nombres de funciones pueden contener letras, dígitos, subrayados y signos de dólar.
- Empezar con un guión bajo _ o letras **a-z** o **A-Z**.
- Contener caracteres en mayúsculas y minúsculas. (Latino es sensible a las mayúsculas y minúsculas, por lo que los identificadores con nombres similares pero con letras mayúsculas o minúsculas en ellas serán interpretadas como diferentes funciones en Latino).

Los nombres de funciones NO pueden:

- No puede existir más de una función con el mismo nombre.
- No son validas las letras acentuadas u otros caracteres como la **ñ**.
- Empezar por un número.
- Empezar por un símbolo o alguna *palabra reservada* de Latino.

Note: En otras palabras los nombres de funciones se rigen por las mismas normas que los nombres de las *variables*.

Múltiples parámetros

Una función puede recibir tantos parámetros como queramos.

Los paréntesis pueden incluir nombres de parámetros y estos parámetros están separados por una **coma (,)**. Ejemplo: (**parámetro1**, **parámetro2**, **parámetro3**, **etc**...)

Al usar más de un parámetro, los valores enviados a la función tienen que estar en el mismo orden que los parámetros asignados en ésta.

5.24.2 Invocar una función

Para ejecutar el código dentro de una función primero hay que invocarla (llamar a la función).

Para llamar una función se hace escribiendo el **nombre de la función** y en paréntesis los parámetros que esta tenga (en caso de que haya alguno).

5.24.3 Retornar función

El retornar los valores de una función nos ayuda a poder procesar de forma independiente valores enviados a esta sin interrumpir nuestro código.

Para que nuestra función pueda retornar valores es necesario que esta tenga sus **parámetros asignados**.

En Latino se puede regresar el valor de una función con las palabras claves **regresar**, **retornar** o la forma corta **ret**. La estructura de una función es la siguiente:

Palabras reservadas

```
regresar
retornar
ret
```

Ejemplo de código

5.24.4 Funciones como variables

Las funciones se pueden usar de la misma manera que las variables, en todos los tipos de fórmulas, asignaciones y cálculos.

5.24.5 Parámetros de funciones

Toda función tiene una **entrada** y una **salida** de datos.

Las salidas de datos se hacen cuando el código de la función se termina de ejecutar y también cuando usamos el comando **retornar**.

Para la entrada de datos en una función hacemos uso de los **parámetros**.

Los parámetros de una función son iguales que las variables con la diferencia que solo funcionan en la función en donde estén declaradas.

5.25 Lista (Arrays)

Las listas o arrays (también son conocidas como arreglos, matrices o vectores en otros lenguajes de programación) son variables que pueden almacenar **múltiples valores** al mismo tiempo y estos a su vez están organizados por **índice**.

Entre algunos de los tipos de datos que una lista puede simultáneamente almacenar pueden ser: lógicos, numéricos, cadenas, otras listas y/o diccionarios.

Note: Latino dispone de una librería para el manejo de listas, *aquí*.

5.25.1 ¿Cúando y/o por qué usar una lista?

Una variable nos permite almacenar únicamente un tipo de dato en ella a la vez. Esto resulta ideal cuando se trabaja con operaciones simples pero no sería practico en grandes operaciones.

Ejemplo:

Si tenemos un grupo de artículos (una lista de marcas de carro, por ejemplo) y las almacenamos en variables, sería algo así:

Sin embargo, ¿qué pasaría si en vez de 3 marcas, tuviéramos 300 marcas y tuviéramos que buscar un valor en específico? Resultaría tedioso y poco practico tener que crear 300 variables con nombres distintos solo para almacenar un valor.

La solución sería usar una lista. Como se había explicado antes, una lista nos permiten almacenar varios valores en ella y acceder a ellos haciendo referencia a su número de índice.

5.25.2 Creación de una lista (array)

Las listas se definen (crean) entre **corchetes [ ]** y sus valores están separados por **comas**.

Las listas o array como vimos en el ejemplo anterior se pueden declarar en una sola línea, pero esto no es el único caso, también podemos declarar una lista en múltiples líneas de la siguiente manera:

5.25.3 Acceder a los valores en una lista

Para acceder a los valores almacenados en una lista usamos su **número de índice**.

Estos números índices comienzan a contar desde el **número 0 (cero)** en adelante.

Índices con números negativos

Podemos utilizar también índices con **números negativos**.

La indexación negativa significa comenzar desde el final, -1 se refiere al último elemento, -2 se refiere al penúltimo elemento, -3 se refiere al antes penúltimo elemento, etc.

Mostrar carácter de un elemento

Anteriormente vimos que usando el número de índice podemos acceder al valor almacenado en una lista, pero también podemos únicamente mostrar un solo carácter de una lista usando doble índice.

Esta propiedad resulta bastante útil cuando se esta buscando palabras con letras específicas o carácteres específicos en los elementos de una lista.

5.25.4 Agregar un nuevo elemento

Para agregar un nuevo elemento a la lista solo basta con escribir el nombre de la lista más el nuevo número de índice.

5.25.5 Cambiar valor de elemento

Se puede cambiar el valor de un elemento con solo hacer referencia a la misma:

5.25.6 Acceder a todos los valores de una lista

Para imprimir todos los valores de una lista solo hacemos mención del nombre de la lista **sin número de índice**.

5.25.7 Las listas pueden ser objetos

Las variables pueden ser objetos, y las listas son un tipo de variables. Debido a esto se puede almacenar diversos tipos de datos en una lista.

Las listas pueden almacenar **funciones**, **otras listas** o **diccionarios** si así se desea.

Anidar una función en una lista

Anidar otras listas

La manera de llamar a una **sub-lista** de una lista es de la misma manera en la que se accede a los **carácteres de un elemento** que explicamos unos puntos más arriba de este articulo.

Anidar un diccionario a una lista

Si tenemos un diccionario anidado en una lista, para llamar un elemento del diccionario, escribimos el nombre de la lista con el número de índice en donde se encuentra el diccionario más la propiedad del diccionario:

5.26 Diccionarios (Objetos)

Los diccionarios u objetos, también son llamados **matrices asociativas**, esto deben su nombre a que son colecciones que relacionan una **propiedad (o llave)** a un valor.

Las *listas (arrays)* almacenan sus valores organizados por índeces, pero este no es el caso de los diccionarios los cuales almacenan sus valores utilizando **corchetes [ ]**.

Los diccionarios son una colección de valores almacenados **sin orden** y **sin índeces**. Esto es así porque los diccionarios se implementan como tablas hash, y a la hora de introducir una nueva propiedad (llave) en el diccionario se calcula el hash de la llave para después poder encontrar la entrada correspondiente rápidamente. Si se modificara su propiedad después de haber sido introducida en el diccionario, evidentemente, su hash también cambiaría y no podría ser encontrado.

Los diccionarios u objetos se declaran (crean) entre **llaves { }** y sus propiedades se escriben **"propiedad" : "Valor"** y cada propiedad con su valor están separadas por **comas**.

En el ejemplo anterior se puede apreciar que los diccionarios realmente son **variables**, pero estas almacenan una mayor cantidad de valores.

Note: Latino dispone de una librería para el manejo de diccionarios, *aquí*.

5.26.1 Definir un diccionario

Los diccionarios u objetos como vimos en el ejemplo anterior se pueden declarar en una sola línea, pero esto no es el único caso, también podemos declarar un diccionario en múltiples líneas de la siguiente manera:

Declaración de un diccionario en una sola línea:

Declaración de un diccionario en múltiples líneas:

5.26.2 Propiedades de un diccionario

En programación al conjunto de **llave:valor** se les llaman propiedades.

Propiedad (llave)	Valor de propiedad
marca	Toyota
modelo	Camry
anio	2011
color	Blanco

5.26.3 Invocar una propiedad (llave)

En Latino existen dos maneras para llamar a una propiedad de un diccionario.

Usaremos el ejemplo anterior como base:

5.26.4 Cambiar valor de propiedad

Se puede cambiar el valor de una propiedad con solo hacer referencia a la misma:

5.26.5 Métodos de un diccionario

Los diccionarios además de ser objetos con sus propiedades, también disponen de **métodos**.

Los métodos son **acciones** que se pueden realizar a un objeto. En otras palabras los métodos en realidad son **funciones** almacenadas en las propiedades del objeto.

5.26.6 Invocar un método

Al igual que las propiedades los métodos se invocan de la misma manera, con la diferencia que estos requiere **paréntesis ( )** al final del nombre.

5.26.7 Anidar diccionarios

Un diccionario no únicamente está compuesto por propiedades y métodos, también de otros diccionarios. A esto se lo llama **diccionario anidado**.

De igual manera podemos anidar otros diccionarios ya existentes a un diccionario:

5.26.8 Librería "dic"

La librería **dic** nos permite obtener más información sobre nuestro diccionario en Latino.

Longitud de un diccionario

Para obtener la Longitud de un diccionario usaremos el siguiente comando **dic.longitud( )**.

Llaves de un diccionario

Para obtener las **llaves (propiedades)** que almacena un diccionario usaremos el siguiente comando **dic.llaves( )**.

Valores de un diccionario

Para obtener los **valores** almacenados en las propiedades de un diccionario usaremos el siguiente comando **dic.valores( )**, también podemos usar este otro comando que de igual manera hace lo mismo **dic.vals( )**.

Agregar nuevas propiedades a un diccionario

Para agregar nuevas propiedades a un diccionario implementamos el siguiente código:

Recorrer un diccionario

Podemos recorrer un diccionario utilizando el *ciclo Desde (For Loop)*.

Éste es un ejemplo de cómo podríamos recorrer un diccionario que este a su vez imprima las llaves y valores almacenados:

Note: Por el momento en Latino 1.2.0 en la librería **dic** funciones como **dic.copiar( )**, **dic.eliminar( )** y **dic.existe( )** no están disponibles. Espere a futuros lanzamientos de Latino para ver sus novedades.

5.27 Condición Si (If)

En programación la **condicional Si (If)** ayuda al programador a ejecutar instrucciones cuando una condición lógica es cumplida.

Esta condicional evalúa una condición y si el resultado es verdadero ejecuta las instrucciones descritas en ella.

Latino tiene las siguientes declaraciones condicionales:

Comandos	Descripción
si	Inicio de la declaración condicional y esta ejecuta un bloque de código si su condición es verdadera.
osi	Esta ejecuta un nuevo bloque de código si la condicional anterior fue falsa.
sino	Esta ejecuta un bloque de código si las codiciones anterios son falsas.
fin	Marca el fin de la condicional lógica.

Las **condiciones lógicas** pueden contener expresiones matemáticas.

Expresión	Descripción
a == b	Es igual
a != b	No es igual
a < b	Menor que
a > b	Mayor que
a <= b	Menor o igual que
a >= b	Mayor o igual que
a <> b	Si es diferente

5.27.1 Condicional "si" (if)

Inicio de la declaración condicional y esta ejecuta un bloque de código si su condición es verdadera.

En esta declaración condicional solo se puede escribir el comando **si** una vez y es solo al principio, así también como el comando **fin** solo al final.

Ejemplo de sintaxis

```
si (condición lógica)
  #Bloque de código
fin
```

Ejemplo de código

5.27.2 Condicional "osi" (else if)

Esta ejecuta un nuevo bloque de código si la condicional anterior fue falsa.

El comando **osi** se puede repetir cuantas veces sean necesarias pero cada nuevo comando **osi** debe llegar una nueva **condición lógica**.

Ejemplo de sintaxis

```
si  condición lógica
  #bloque de código
osi condición lógica
  #bloque de código
fin
```

Ejemplo de código

5.27.3 Condicional "sino" (else)

Esta ejecuta un bloque de código si las codiciones anterios son falsas.

El comando **sino** solo se puede escribir una sola vez y debe ir después del comando **si** o después del comando **osi**.

El comando **sino** a diferencia del comando **si** y del comando **osi** no lleva **condición lógica**.

Ejemplo de sintaxis

```
si (condición lógica)
  #bloque de código
sino
  #bloque de código
fin
```

Ejemplo de código

5.27.4 Condicional "si" lineal

Latino al ser un lenguaje de consola, su código se puede escribir en una sola línea, asi:

Este ejemplo también aplica para el comando **osi** y **sino**.

5.27.5 Anidar condición "si"

Una **condicional si** puede tener anidada **otra condición si**, si fuera necesario.

Esta nueva condición anidada puede ir en cualquier bloque de código, dentro del **si**, o del **osi** o también del **sino**

5.27.6 Operadores condicionales

En capítulos pasados ya habiamos hablado de los *operadores condicionales*. Aquí volveremos a repasar esta clase de operador.

El operadore condicional es puede implementar como una alternativa de la **condicional si**, ya que requiere de menos líneas de código para ejecutar su condicional lógica.

Sintaxis del operador condicional:

```
(Expresión condicional) ? expresión1 : expresión2
```

El operador condicional funciona de la siguiente manera:

- La primera *expresión condicional* se evalúa primero. Esta expresión se evalúa si es verdadera o si es falsa.
- Si la expresión condicional es verdadera, se evalúa la *expresión1*.
- Si la expresión condicional es falsa, se evalúa la *expresió2*.

5.27.7 Operadores lógicos

En capítulos pasados vimos que los *operadores lógicos* son el **Y lógico**, el **Ó lógico** y el **No lógico**.

Estos al igual que las expresiones matemáticas también se pueden utilizar en la **condicional si**.

"Y" lógico

El **Y lógico** se expresa así: **&&**

"Ó" lógico

El **Ó lógico** se expresa así: **||**

"No" lógico

El **No lógico** se expresa así: **!**

5.28 Condición Elegir (Switch)

La **condicional elegir** es una alternativa a la *condicional si* que vimos en el capítulo anterior.

Esta condicional es usada para realizar diferentes acciones basándose en diferentes condiciones lógicas. En otras palabras, esta condicional evalúa una opción en múltiples casos posibles y selecciona uno de varios bloques de códigos para ser ejecutados.

La condicional elegir contiene las siguientes declaraciones:

Comandos	Descripción
elegir	Inicio de la declaración. Esta evalúa la expresión condicional.
caso	El resultado de la expresión es evaluada y si concuerda con cualquier caso este es ejecutado.
defecto	En caso de no producirse ninguna concordancia en ninguno de los casos, este bloque de código será ejecutado.
otro	Es exactamente lo mismo que el comando **defecto** solo con otro nombre.
fin	Marca el fin de la declaración.
romper	Detiene la ejecución del código.

5.28.1 Sintaxis de la condicional "elegir"

Ejemplo de sintaxis

```
elegir(expresión)
  caso 1:
    #Bloque de código
    romper
  caso 2:
```

(continues on next page)

(continued from previous page)

```
    #Bloque de código
    romper
  defecto:
    #Bloque de código
fin
```

Ejemplo de código

Note: Notese que se tiene que repetir el caso para B y C por el momento no se permiten casos múltiples como una sola opción.

5.29 Condición Desde (For Loop)

La **condicional desde** hace repetir un mismo código una y otra vez hasta que su expresión sea cumplida (sea verdadera).

Regularmente, la condicional **desde** se utiliza para navegar entre los elementos de una **lista** o **diccionario**, pero también para ejecutar códigos que seán repetitivos.

5.29.1 Sintaxis de la condicional "desde"

Ejemplo de sintaxis

```
desde (declaración; expresión; sentencia)
  #Bloque de código
fin
```

Declaraciones	Descripción
Declaración	Esta se ejecuta (una sola vez) antes de la ejecución del código
Expresión	Define las condiciones para que el bloque de código sea ejecutado
Sentencia	Esta se ejecuta (cada vez) después de ejecutar el bloque de código

Ejemplo del código

Se puede especificar un salto diferente cambiando la expresión de incremento.

5.29.2 Uso de librerías

Las **expresiones** de la condicional **desde** pueden implementar el uso de librerías, así:

5.29.3 Anidar condicional "desde"

Cuando una condicional **desde** contiene otro condicional **desde** dentro de ella, se le llama **condicional anidada**.

Cuando la condicional "desde" (**madre**) se ejecuta y llega a una condicional anidada (**hija**), la condicional madre no continúa hasta que la condicional hija termine **todos sus ciclos** (hasta que sea verdadera). Todo este ciclo se repetirá hasta que la condicional madre sea verdadera.

5.30 Condición Mientras (While Loop)

La **condicional mientras** ejecuta un bloque de código repetidas veces mientras su **condición** se siga cumpliendo (sea verdadera).

Esta condicional primero verifica que su condición se cumpla antes de ejecutar el código.

5.30.1 Sintaxis de la condicional "mientras"

Ejemplo de sintaxis

```
mientras (condición)
  #Bloque de código
fin
```

Ejemplo de código

Note: En este ejemplo, si no se incrementa el valor de la variable **i** este bucle se repetiría infinitamente sin parar, produciendo un error en la memoria.

5.30.2 Diferencias entre "mientras" y "desde"

Como habrá notado, el **comando mientras** y el **comando desde** son muy parecidos entre sí, con la diferencia que el comando **mientras** no requiere de una **declaración** o **sentencia** como en el comando **desde**.

Ejemplo #1: "desde"

En este ejemplo se utiliza el comando **desde** para adquirir las marcas de carros en una variable.

Ejemplo #2: "mientras"

Este ejemplo es similar al anterior con la diferencia que se utilizara el **comando mientras**.

5.31 Condición Repetir (Do While)

La **condicional repetir** es una variante de la **condicional mientras**.

Esta condicional ejecuta su bloque de código al menos una vez antes de verificar si su condición es cumplida (es verdadera).

Mientras su condición sea verdadera, este bucle se volverá a repetir.

5.31.1 Sintaxis de la condicional “repetir”

Ejemplo de sintaxis

```
repetir
  #Bloque de código
hasta (condición)
```

Note: En este ejemplo, si no se incrementa el valor de la variable **i** este bucle se repetiría infinitamente sin parar, produciendo un error en la memoria.

5.32 Módulos

Un módulo es cualquier archivo o directorio en donde se divide el programa en muchos componentes o bloques de códigos autónomos que pueden ser utilizados en diferentes partes del programa.

5.32.1 Creación de módulos

Un módulo no es más que otro archivo en Latino, el cual puede contener desde *variables*, *funciones*, *listas*, *diccionarios*, etc.

Para crear un módulo en Latino necesitamos guardar el código deseado en un archivo con extensión **.lat**.

Además es necesario definir en un *diccionario* las variables y funciones que serán exportadas desde el módulo con el comando **retornar**.

sintaxis de un modulo

```
variableMensaje = "Hola mundo"

retornar {"comandoAExportar":variableMensaje}
```

5.32.2 Uso de módulos

Para usar un módulo necesitamos utilizar el comando *incluir( )*.

Cuando se importan un módulo en Latino es necesario asignarlo a una variable, de esta manera el programa sabrá en cuál módulo buscar la función deseada.

Ejemplo con módulo

Para este ejemplo usaremos dos archivos los cuales deben de estar en la misma ubicación de carpeta.

Código Principal

Código del módulo

El primer archivo será nuestro código principal en el cual utilizaremos el comando **incluir( )** y después usaremos las funciones importadas del módulo.

El segundo archivo será el código de nuestro módulo, el cual guardaremos con el nombre de **moduloPersona.lat**.

Important: Es necesario especificar en el comando **retornar** las funciones y variables que deseamos que se exporten de este módulo, de lo contrario el módulo no funcionara.

5.32.3 Subdirectorio

Con el comando **incluir( )**, para especificar archivos en subdirectorios varia dependiendo del sistema operativo.

En MS-Windows, para especificar un archivo en un subdirectorio usamos \.

En Linux y Mac, para especificar un archivo en un subdirectorio usamos /.

Ejemplo

Para añadir un archivo del siguiente subdirectorio **carpeta/modulo** lo especificamos de la siguiente manera:

```
incluir("carpeta\modulo")      //MS-Windows
incluir("carpeta/modulo")      //Unix
```

5.33 RegEx

Una expresión regular (*regular expression* ó *RegEx*, por su nombre en Ingles) es una secuencia de caracteres que forman un patrón de búsqueda.

Una RegEx puede ser utilizada para comprobar si una cadena(string) contiene un patrón de carácter específico.

Para hacer uso de las expresiones regulares se pueden utilizar los siguientes comandos:

- **~=** de este operador relacional también se hace mención en *este artículo*.
- **cadena.regexl( )** el cual devuelve un valor buleano.
- **cadena.regex( )** el cual devuelve una lista de todas las coincidencias.

5.33.1 Operador ~=

Ejemplo de sintaxis

```
~= (expresión)
```

5.33.2 cadena.regexl( )

Ejemplo de sintaxis

```
cadena.regexl(texto, expresión)
```

5.33.3 cadena.regex( )

Ejemplo de sintaxis

```
cadena.regex(texto, expresión)
```

Ejemplo 1

Ejemplo 2

Ejemplo 3

5.33.4 Metacaracteres

Los metacaracteres son caracteres con un significado especial.

Carácter	Descripción	Ejemplo
[ ]	Conjunto de caracteres	'[a-m]'
\	Señaliza una secuencia especial	'\d'
.	Cualquier carácter(excepto nuevas líneas)	'Bien..dos'
^	Comienza con...	'^hola'
$	Termina con...	'mundo$'
*	Cero o más caracteres	'Bienve*'
+	Uno o más caracteres	'Hol+'
{ }	Número especifico de caracteres	'Bien{7}'
|	Cualquiera de las opciones	'x|y'
( )	Grupo	

5.33.5 Conjuntos

Los conjuntos son caracteres dentro de **corchetes [ ]** los cuales tiene un significado especial.

Conjunto	Descripción
[arn]	Devuelve los conjuntos donde estén presentes cualquiera de los caracteres especificados (**a**, **r** ó **n**)
[a-n]	Devuelve los conjuntos de cualquier letra minúscula que estén alfabeticamente entre **a** y **n**
[^arn]	Devuelve cualquier conjunto de caracteres EXCEPTO **a**, **r** y **n**
[0123]	Devuelve los conjuntos donde estén presentes cualquiera de los digitos especificados (**0**, **1**, **2** ó **3**)
[0-9]	Devuelve cualquier conjuntos de digitos entre **0** y **9**
[0-5][0-9]	Devuelve los conjuntos de cualquiera de los dos digitos entre **00** y **59**
[a-zA-Z]	Devuelve cualquier conjunto de caracteres desde la **a** hasta la **z** sean minúsculas o mayusculas
[+]	Los conjuntos **+**, *****, **.**, **|**, **( )**, **$**, **{ }** no tienen un significado especial. [+] significa que devolverá una coincidencia para cualquier carácter **+** en la cadena.

5.33.6 secuencias especiales

Una secuencia especial inicia con una **barra invertida ** seguido de uno carácter de la siguiente lista, los cuales tienen un significado especial.

Carácter	Descripción
\A	Devuelve un conjunto si el carácter especificado está al inicio del texto
\b	Devuelve un conjunto cuando un carácter especificado está al inicio o al final de una palabra
\B	Devuelve un conjunto cuando los caracteres especificados están presentes pero NO al inicio o al final de la palabra
\d	Devuelve un conjunto cuando la cadena(string) contiene números
\D	Devuelve un conjunto cuando la cadena(string) NO contiene números
\s	Devuelve un conjunto cuando la cadena(string) contiene almenos un espacio en blanco
\S	Devuelve un conjunto cuando la cadena(string) NO contiene espacios blanco
\w	Devuelve un conjunto cuando la cadena(string) cualquier carácter(a_Z, 0-9 ó _)
\W	Devuelve un conjunto cuando la cadena(string) NO contiene ningún carácter
\Z	Devuelve un conjunto si el carácter especificado está al final de la cadena(string)

5.33.7 Patrones de expresiones regulares

Carácter [ ]

Los brackets son utilizados para buscar caracteres en un rango asignado.

Expresión	Descripción
[abc]	Busca cualquier carácter asignado en los brackets
[0-9]	Busca cualquier número asignado en los brackets
(x|y)	Busca cualquiera de las alternativas separados por '|'

5.34 acadena( )

La función **acadena( )** convierte los números a cadenas(textos).

Esta función puede ser utilizada en cualquier tipo de números, decimales, variables, o expresiones.

```
acadena(x)            //Devolverá como texto el valor numérico de la variable X
acadena(123)          //Devolverá como texto el valor numérico 123
acadena(100 + 23)     //Devolverá como texto el resultado de la expresión
```

Ejemplo de función

5.34.1 Convertir buleanos a cadenas(textos)

La función **acadena( )** se puede usar para convertir valores buleanos(lógicos) en cadenas(textos).

5.35 alogico( )

La función **alogico( )** convierte los números y textos a un valor lógico (**verdadero** o **falso**).

Cualquier número diferente a **0**, ya sea un número positivo, negativo o decimal, dará **verdadero**.

De igual manera cualquier texto aun si este es un espacio en blanco devolverá **verdadero**.

La función **alogico( )** devolverá un valor **falso** solo si el número es **0**, si no hay nada en las comillas, o si asignamos un valor **nulo**.

Ejemplo de función

5.36 anumero( )

La función **anumero( )** convierte las cadenas(textos) a números.

Las comillas sin caracteres, ejemplo **‘’** devolverá un valor **Nulo** al igual que el números **0**.

Los espacios en blanco, ejemplo **‘ ‘** o cualquier número que tengo un espacio en blanco, ejemplo **‘7 ‘** estos serán convertidos su valor de código ALT.

Ejemplo de función

5.36.1 Convertir buleanos a número

La función **anumero( )** se puede usar para convertir valores buleanos(lógicos) en un valor numérico.

5.37 imprimir, escribir, poner( )

Las funciones **imprimir( )**, **escribir( )**, y **poner( )** escriben en pantalla la cadena, número, o expresión asignadas en ellas.

Estas funciones **imprimir( )**, **escribir( )**, y **poner( )** son literalmente la misma función, pero con distintos nombres.

La razón de la amplia variedad de nombres que recibe esta función es por las diferentes maneras que existen en otros lenguajes de programación para realizar la misma función. Ejemplo: algunos lenguajes utilizan el comando **print** y otros el comando **put**.

Ejemplo de función

5.38 imprimirf( )

El comando **imprimirf( )** esta inspirado en el comando **printf( )** en C, pero el comando en Latino es mas limitado.

Este comando en esencia es similar a los comandos **imprimir( )**, **escribir( )** y **poner( )** pero con algunas diferencias.

El comando **imprimirf( )** requiere del carácter **\n** al final de cada cadena para poner **escribir** en pantalla, caso que NO se da con los otros comandos.

Este comando permite **dar formato** a un carácter o valor ASCII.

El comando **imprimirf( )** opera con los siguientes formatos:

- **%c**, convierte a un carácter el valor ASCII.
- **%i**, convierte a un número enteros.
- **%f**, convierte a un número decimal.
- **%d**, convierte a un número.
- **%o**, convierte a un valor octal.
- **%x**, convierte a un hexadecimal.
- **%e**, convierte a una expresión científica.
- **%s**, convierte a carácter o ha una cadena de texto.
- **%%**, Devuelve el simbolo de **porcentage (%)**.

5.39 incluir( )

La función **incluir( )** nos permite importar **módulos** y **librerías** a nuestro proyecto.

Cuando escribimos el módulo o librería, este debe ser escrito entre **comillas** y no es necesario escribir su extensión.

Ejemplo de sintaxis

Note: Esta función solo puede incluir archivos con extensión **.lat** o archivos de librerías que hayan sido escritas para Latino y que usen su API.

5.39.1 Subdirectorio

Con el comando **incluir( )**, para especificar archivos en subdirectorios varia dependiendo del sistema operativo.

En MS-Windows, para especificar un archivo en un subdirectorio usamos ****.

En Linux y Mac, para especificar un archivo en un subdirectorio usamos **/**.

Ejemplo

Para añadir un archivo del siguiente subdirectorio **carpeta/modulo** lo especificamos de la siguiente manera:

```
incluir("carpeta\modulo")      //MS-Windows
incluir("carpeta/modulo")      //Unix
```

5.40 leer( )

La función **leer( )** escanea las teclas numéricas y alfanuméricas digitadas por el usuario, hasta que este presione la tecla **enter**.

Es recomendable asignar este comando a una variable, ya que se puede manipular con mayor facilidad cualquier dato digitado por el usuario.

```
leer()
```

5.41 limpiar( )

La función **limpiar( )** limpia la pantalla de la consola.

Esta función es exactamente igual como si usáramos **cls** en el CMD de MS-Windows o como utilizar **clear** en sistemas basados en UNIX.

```
limpiar()
```

5.42 tipo( )

La función **tipo( )** devuelve el tipo de valor almacenado en una variable o memoria.

Los valores que esta función nos puede devolver lo podemos encontrar en *tipos de datos*.

5.43 Lib "archivo"

La librería **archivo** contiene las funciones principales para el manejo de archivos en Latino.

Cada uno de estos comandos puede recibir el **nombre** como también la **ruta** del archivo.

El nombre de archivo o ruta del archivo deben ser escritas entre **comillas**.

Important: Al marcar las rutas es importante utilizar doble slash \ \ en MS-Windows

Y en macOS o Linux utilizar slash invertido /

Lista de comando

Comando	Parámetros	Descripción
anexar( )	2	Agrega un texto o dato al final del archivo
borrar( )	1	Elimina el archivo especificado
eliminar( )		
crear( )	1	Crea un archivo con el nombre especificado
duplicar( )	2	Hace un duplicado del archivo especificado
ejecutar( )	1	Ejecuta el archivo especificado
escribir( )	2	Escribe y/o Sobrescribe en el archivo
leer( )	1	Lee el contenido de un archivo y lo convierte en cadena
lineas( )	1	Almacena en una lista cada línea del archivo
renombrar( )	2	Renombra un archivo por un nuevo nombre asignado

5.43.1 archivo.anexar( )

Este comando nos permite **agregar** texto al final del documento especificado.

A diferencia del comando *archivo.escribir( )* que sobrescribe los datos existentes en el documento, el comando **archivo.anexar( )** añade el texto al final del documento.

5.43.2 archivo.duplicar( )

Este comando crea un **duplicado** de un archivo especificado.

Ejemplo de sintaxis

```
archivo.duplicar("archivo_Original", "archivo_Copia")
```

5.43.3 archivo.crear( )

Este comando nos permite **crear un archivo** con un nombre especificado en cualquier ruta de nuestro sistema.

Note: Si al especificar la ruta del archivo a crear escribimos un nombre de alguna carpeta que no existe, este no hará nada, ya que este comando solo puede crear archivos y no carpetas.

5.43.4 archivo.ejecutar( )

Este comando nos permite la **ejecución** de un archivo que contenga código de Latino.

5.43.5 archivo.eliminar( )

Este comando nos ayuda a **eliminar** un archivo especificado.

5.43.6 archivo.escribir( )

Este comando nos permite **escribir** y **sobrescribe** un archivo especificado.

Important: Si deseamos añadir más información al archivo usar el comando *archivo.anexar( )*

Si se usa este comando en un archivo **NO vacío** este será completamente **sobrescribe** con la nueva información.

5.43.7 archivo.leer( )

Para este comando se requiere **almacenar en una variable** el contenido del archivo que deseamos leer.

5.43.8 archivo.lineas( )

Este comando almacena en una **lista** cada línea de código de un archivo especificado.

Para este comando es requerido asignarlo a una variable para almacenar el contenido del archivo.

5.43.9 archivo.renombrar( )

Este comando nos permite **renombrar** el nombre de un archivo.

Este comando también adminte rutas.

Ejecuta de sintaxis

```
archivo.renombrar(Nombre_viejo, Nombre_nuevo)
```

5.44 Lib "cadena"

La librería **cadena** nos permite trabajar y manipular las *cadenas (string)* en Latino.

Lista de comando

Comando	Parámetros	Descripción
bytes( )	1	Devuelve el valor ASCII de cada carácter en una lista
char( )	1	Combierte el valor o lista ASCII a un carácter o una lista de caracteres

Continued on next page

Table 1 – continued from previous page

Comando	Parámetros	Descripción
comparar( )	2	Compara dos cadenas de textos y devuelve un valor númerico
concatenar( )	2	Combina dos cadenas de textos en una sola cadena
contiene( )	2	Devuelve un valor buleano si encuentra la palabra o cadena especificada
ejecutar( )	1	Ejecuta una cadena que tenga código de latino
eliminar( )	2	Elimina la primera considencia en una cadena
encontrar( )	2	Regresa la posición de la primera considencia encuentra
indice( )		
es_alfa( )	1	Comprueba si la cadena solo contiene texto y/o números
es_igual( )	2	Regresa verdadero si las dos cadenas son iguales
es_numerico( ) es_numero( )	1	Comprueba si la cadena solo contiene números
esta_vacia( )	1	Regresa verdadero si la cadena está vacia
formato( )	1	Asigna un formato a un carácter
inicia_con( )	2	Comprueba si la cadena inicia con un texto o cadena especificado
insertar( )	3	Agrega una cadena en la posición indicada
invertir( )	1	Invierte el contenido de la cadena
longitud( )	1	Regresa el tamaño de la cadena
mayusculas( )	1	Combierte toda la cadena en mayúsculas
minusculas( )	1	Combierte toda la cadena en minúsculas
recortar( )	1	Elimina los espacios al inicio y al final de la cadena
reemplazar( )	4	Cambiar una palabra por otra en una cadena
regex( )	2	Utiliza RegEx y regresa una lista de las coincidencias
regexl( )	2	Regresa un valor buelano si encuentra la coincidencia
rellenar_derecha( )	3	Agrega n caracteres al final de la cadena especificada
rellenar_izquierda( )	3	Agrega n caracteres al inicio de la cadena especificada
separar( )	2	Separa la cadena en una lista en base a un separador
subcadena( )	3	Devuelve una sub-cadena en base a la posición y su longitud
termina_con( )	2	Devuelve un valor buleano si hay una coincidencia
ultimo_indice( )	2	Devuelve la posición final de un caracter o palabra especificada

5.44.1 cadena.bytes( )

Este comando nos permite **convertir** nuestra **cadena de texto** a valor ASCII y la devuelve en una lista.

El comando **cadena.char( )** es su contraparte, ya que convierte los valores ASCII a textos.

5.44.2 cadena.char( )

Este comando nos permite **convertir** un número o **lista de** valores ASCII a una cadena de texto.

El comando **cadena.bytes( )** es su contraparte, ya que convierte los textos a valores ASCII.

5.44.3 cadena.comparar( )

Este comando **comparar** dos cadenas de textos **carácter por carácter** hasta encontrar la primera diferencia.

Este comando es similar al comando **strcmp()** en C.

El comando **cadena.comparar( )** devuelve los siguientes valores:

- -1, si la primera cadena es **menor** que la segunda.
- 0, si ambas cadenas son **iguales**.
- 1, si la primera cadena es **mayor** que la segunda.

Note: **menor**, **igual** o **mayor** hacen referencia al orden o posición de las letra en el alfabeto.

5.44.4 cadena.concatenar( )

Este comando nos permite **unir** dos cadenas de textos en una sola.

El comando **cadena.concatenar( )** es una alternativa al comando **doble punto (..)**.

5.44.5 cadena.contiene( )

Este comando nos permite **verificar** si existe una **coincidencia** del texto o cadena a buscar en otra y devolverá un valor buleano.

5.44.6 cadena.ejecutar( )

Este comando nos permite **ejecutar** una cadeta de texto de tenga código de Latino.

5.44.7 cadena.eliminar( )

Este comando solo **elimina la primera coincidencia** encontrada en una cadena de texto.

5.44.8 cadena.encontrar( )

Este comando **busca** la posición de la primera coincidencia de caracteres o textos.

Este comando también dispone de un alias **cadena.indice( )**.

El comando **cadena.encontrar( )** cuenta cada carácter de una cadena de texto hasta encontrar la primera coincidencia.

El comando comienza a contar desde el número **cero (0)** como primer número en adelante.

Si el texto o cadena no fue encontrado, entonces devolverá **-1**.

5.44.9 cadena.es_alfa( )

Este comando **comprueba** si la cadena solo contiene valores **alfanuméricos** y NO símbolos.

El comando **cadena.es_alfa( )** devolverá un valor buleano:

- **verdadero** si la cadena es letras y/o números.
- **falso** si la cadena contiene o es un símbolo.

5.44.10 cadena.es_igual( )

Este comando **comprueba** si ambas cadenas **coinciden entre sí** y regresa un valor buleano.

5.44.11 cadena.es_numero( )

Este comando **comprueba** si la cadena **solo contiene números** y devolverá un valor buleano.

Este comando también dispone de un alias **cadena.es_numerico( )**.

5.44.12 cadena.esta_vacia( )

Este comando **verificar** que la cadena está vacía.

El comando **cadena.esta_vacia( )** devolverá un valor buleano:

- **verdadero** si la cadena esta vacía.
- **falso** si la cadena NO esta vacía.

5.44.13 cadena.formato( )

Este comando permite **dar formato** a un carácter o valor ASCII.

Este comando es similar al comando **imprimirf( )**, aunque este ultimo requiere del carácter **\n** para poder escribir en pantalla.

El comando **cadena.formato( )** opera con los siguientes formatos:

- **%c**, convierte a un carácter el valor ASCII.
- **%i**, convierte a un número enteros.
- **%f**, convierte a un número decimal.
- **%d**, convierte a un número.
- **%o**, convierte a un valor octal.
- **%x**, convierte a un hexadecimal.
- **%e**, convierte a una expresión científica.

- **%s**, convierte a carácter o ha una cadena de texto.
- **%%**, Devuelve el simbolo de **porcentage (%)**.

5.44.14 cadena.inicia_con( )

A diferencia del comando **cadena.termina_con( )**, este comando **comprueba** si la cadena de texto **inicia con** un carácter especificado, y este devolverá un valor buleano.

Este comando distingue entre **mayúsculas** y **minúsculas**.

5.44.15 cadena.insertar( )

Este comando nos permite **añadir** una cadena a otra cadena de texto en cualquier posición especificada.

La posición se maneja contando cada carácter de la cadena original. Este conteo inicia desde el número **cero (0)** como primer número en adelante.

Ejemplo de sintaxis

```
cadena.insertar(cadena_original, cadena_a_agregar, la_posición)
```

5.44.16 cadena.invertir( )

Este comando nos permite **invertir** el orden de la cadena.

5.44.17 cadena.longitud( )

Este comando retorna la **longitud** de la cadena en dígitos.

El comando comienza a contar desde el número **uno (1)** como primer número en adelante.

5.44.18 cadena.mayusculas( )

Este comando nos permite **transformar** toda nuestra cadena a letras **mayúsculas**.

5.44.19 cadena.minusculas( )

Este comando nos permite **transformar** toda nuestra cadena a letras **minúsculas**.

5.44.20 cadena.recortar( )

Este comando **elimina** cualquier **carácter de espacio** al inicio y al final de la cadena, ya sea espacio en blanco o tabulación.

5.44.21 cadena.reemplazar( )

Este comanod nos permite **cambiar** una palabra por otra en una cadena

Ejemplo de sintaxis

```
(cadena_original, texto_a_reemplazar, texto_nuevo, posición)
```

Note: Este comando cambia el texto seleccionado por el nuevo texto asignado, **mas no lo guarda**.

Para guardar el cambio es recomendable asignarlo a una variable.

5.44.22 cadena.regex( )

Este comando hace uso de las **Expresiones Regulares** o **RegEx** para hacer una **búsqueda avanzada** y retorna una lista con cada una de las coincidencias.

Para aprender más sobre este comando y las expresiones regulares, mire el artículo de RegEx, *aquí*.

5.44.23 cadena.regexl( )

Este comando es conocido como **regex lógico**.

Este comando hace use de las **Expresiones Regulares** o **RegEx** para hacer una **búsqueda avanzada** y retorna **verdadero** si encuentra la coincidencia y **falso** si no la encontró.

Para aprender más sobre este comando y las expresiones regulares, mire el artículo de RegEx, *aquí*.

5.44.24 cadena.rellenar_derecha( )

Este comando nos permite **añadir al final** de la cadena especificada un texto o cadena.

El comando **cadena.rellenar_derecha**( ) nos permite indicar la cantidad de veces que deseamos se repita el nuevo texto a añadir.

Ejemplo de sintaxis

```
cadena.rellenar_derecha(cadena_original, cadena_a_agregar, long_cadena_original +␣
↪cantidad_de_repeticiones(Valor número))
```

5.44.25 cadena.rellenar_izquierda( )

Este comando nos permite **añadir al inicio** de la cadena especificada un texto o cadena.

El comando **cadena.rellenar_izquierda( )** nos permite indicar la cantidad de veces que deseamos se repita el nuevo texto a añadir.

Ejemplo de sintaxis

```
cadena.rellenar_izquierda(cadena_original, cadena_a_agregar, long_cadena_original +␣
↪cantidad_de_repeticiones(Valor numérico))
```

5.44.26 cadena.separar( )

Este comando nos permite **segmentar** una cadena de texto al especificar un separador y el resultado lo devuelve en una lista.

El separador debe ser especificado **dentro de comillas**.

Si no se le asigna un separador, por defecto buscara los espacios en blanco.

Ejemplo de sintaxis

```
cadena.separar(cadena_original, separador)
```

5.44.27 cadena.subcadena( )

Este comando **copia** de una cadena el texto deseado el cual se define indicando **en donde inicia** y la **longitud** que deseamos que tenga el texto a copiar.

La **posición_inicial** comienza a contar desde el número **cero (0)** en adelante.

La **longitud** comienza a contar desde el número **uno (1)** en adelante.

Ejemplo de sintaxis

```
cadena.subcadena(cadena_original, posición_inicial(número), longitud(número))
```

5.44.28 cadena.termina_con( )

A diferencia del comando **cadena.inicia_con**, este comando nos permite **buscar** en una cadena de texto si esta **termina con** un carácter especificado y devuelve un valor buelano.

Este comando distingue entre **mayúsculas** y **minúsculas**.

5.44.29 cadena.ultimo_indice( )

Este comando devuelve la **última posición encontrada** del carácter especificado.

Este comando comienza a contar desde el número **cero (0)** en adelante.

5.45 Lib “dic”

La librería **dic** nos permite trabajar y manipular los *diccionarios* en Latino.

Lista de comando

Comando	Parámetros	Descripción
contiene( )	2	Devuelve verdadero si el elemento existe
eliminar( )	2	Elimina la llave asignada y su valor
llaves( )	1	Devuelve el nombre de la propiedad (llave)
longitud( )	1	Devuelve la longitud de texto
valores( )	1	Devuelve el valor de la propiedad
vals( )		

5.45.1 dic.contiene( )

Este comando nos permite comprobar si una **llave** o **propiedad** existe en un diccionario.

Este comando es sensible a las mayúsculas y minúsculas.

5.45.2 dic.eliminar( )

Este comando nos permite **eliminar** una **llave** o **propiedad** existente de un diccionario.

Este comando es sensible a las mayúsculas y minúsculas.

5.45.3 dic.llaves( )

Para obtener las **llaves (propiedades)** que almacena un diccionario usaremos el siguiente comando **dic.llaves( )**.

5.45.4 dic.longitud( )

Para obtener la Longitud de un diccionario usaremos el siguiente comando **dic.longitud( )**.

5.45.5 dic.valores( )

Para obtener los **valores** almacenados en las propiedades de un diccionario usaremos el siguiente comando **dic.valores( )**, también podemos usar este otro comando que de igual manera hace lo mismo **dic.vals( )**.

5.46 Lib "lista"

La librería **lista** contiene las funciones para el manejo de *lista* en Latino.

Lista de comando

Comando	Parámetros	Descripción
agregar( )	2	Agrega un elemento al final de la lista
comparar( )	2	Comprueba el orden y tamaño de letras de los elementos
concatenar( )	2	Uné todos los elementos de dos listas en una sola
contiene( )	2	Devuelve verdadero si el elemento existe en la lista
crear( )	1	Crea una lista con el nombre especificado
eliminar( )	2	Elimina solo la primera coincidencia de la lista
eliminar_indice( )	2	Elimina un elemento de la lista por posición
encontrar( )	2	Devuelve el índice del elemento buscado
indice( )		
extender( )	2	Agrega los elementos de una lista en otra lista
insertar( )	3	Inserta un nuevo elemento a una lista
invertir( )	1	Invierte el orden de la lista
longitud( )	1	Devuelve la cantidad de elementos de una lista
separador( )	2	Separa los elementos de una lista

5.46.1 lista.agregar( )

Para agregar un nuevo elemento a la lista usamos el comando **lista.agregar()**.

Este comando agrega un nuevo elemento al final de la lista.

5.46.2 lista.comparar( )

Con este comando podremos **comparar la longitud** de dos listas.

Este comando devolverá los siguientes valor según el resultado:

- -1 si la lista original es menor.
- 1 si la lista original es mayor.
- 0 si ambas listas son iguales.

Ejemplo de sintaxis

```
lista.comparar(listaOriginal, listaAComparar)
```

5.46.3 lista.concatenar( )

Con este comando podremos unir **dos listas** en una **nueva lista**.

5.46.4 lista.contiene( )

Este comando nos permite comprobar si un elemento **existe en la lista**.

Este comando es sensible a las mayúsculas y minúsculas.

5.46.5 lista.crear( )

Este comando nos permite crear una lista asignando la **cantidad de elementos** que esta tendrá.

Al crear una lista con este comando, cada elemento tendrá un valor **nulo** los cuales podrán ser modificados después.

Este comando admite un valor número positivo, de ser asignado cualquier número negativo, esta creará una lista vacía.

5.46.6 lista.eliminar( )

A diferencia de **lista.eliminar_indice( )** el comando **lista.eliminar( )** elimina la primera coincidencia de la lista.

5.46.7 lista.eliminar_indice( )

Este comando nos permite **eliminar** un elemento de la lista asignada por medio de su **número de índice**.

En una lista los elementos de esta están organizados por índices y estos índices inicial desde el número **cero (0)** en adelante.

Este comando NO admite números negativos.

5.46.8 lista.indice( )

El comando **lista.indice( )** también dispone de un alias el cual es **lista.encontrar( )**. Este comando nos permite **buscar** un elemento por su nombre en una lista y nos devolverá su **número de índice**.

Este comando es sensible a las mayúsculas y minúsculas.

Si NO encuentra el nombre del elemento a buscar, entonces regresara **-1**.

5.46.9 lista.extender( )

El comando **lista.extender( )** copiará los elementos de una lista para ser insertados al final de otra lista deseada.

Ejemplo de sintaxis

```
lista.extender("ListaAExtender", "ListaACopiar")
```

5.46.10 lista.insertar( )

Con este comando podemos insertar un elementos en cualquier indice deseado de una lista.

Ejemplo de sintaxis

```
lista.insertar(listaOriginal, elementoNuevo, indice)
```

5.46.11 lista.invertir( )

Para invertir el orden de una lista, utilizamos el comando **lista.invertir( )**.

5.46.12 lista.longitud( )

Este comando devuelve la cantidad de elementos de una lista.

5.46.13 lista.separar( )

El comando **lista.separar( )** nos permite separar cada elemento de la lista con un separador asignado.

El separador debe ser declarado dentro de comillas.

Por defecto si no se indica un separador este será sustituido por un espacio en blanco.

Ejemplo de sintaxis

```
lista.insertar(lista, separador)
```

5.47 Lib "mate"

La librería **mate** contiene las funciones de **matemáticas** en Latino.

Lista de comando

Comando	Parámetros	Descripción
abs( )	1	Devuelve el valor absoluto
acos( )	1	Devuelve el arcocoseno en radianes
acosh( )	1	Devuelve el coseno hiperbólico inverso de un número
aleatorio( )	0, 1 ó 2	Devuelve un número aleatorio
alt( )		
asen( )	1	Devuelve el arcoseno en radianes
asenh( )	1	Devuelve el arcoseno hiperbólico inverso de un número
atan( )	1	Devuelve el arcotangente como un valor numérico entrew -PI/2 y PI/2 radianes
atanh( )	1	Devuelve el arcotangente hiperbólico inverso de un número
atan2( )	2	Devuelve el arcotangente del cociente de sus argumentos

Continued on next page

Table 2 – continued from previous page

Comando	Parámetros	Descripción
base( )	2	Devuelve la base de la operación
cos( )	1	Devuelve el coseno
cosh( )	1	Devuelve el coseno hiperbólico
e( )	0	Devuelve el valor del número de Eular (Eular's number)
exp( )	1	Devuelve el valor de E^x, donde E es un número de Euler
frexp( )	2	Devuelve el número descompuesto y a una potencia integral de dos.
ldexp( )	2	Devuelve el número multiplicado por 2 elevado a una potencia
log( )	1	Devuelve el logaritmo natural
log10( )	1	Devuelve el logaritmo natural en base diez
parte( )	2	Devuelve la parte de la operación
pi( )	0	Devuelve el valor de PI
piso( )	1	Devuelve el número redondeado hacia abajo al número entero más cercano
porc( )	2	Devuelve el porcentaje de la op-
porciento( )		eración
porcentaje( )		
pot( )	2	Devuelve el valor de un número elevado a la potencia
max( )	1,2,3...	Devuelve el más alto valor de una lista
min( )	1,2,3...	Devuelve el más bajo valor de una lista
raiz( )	1	Devuelve la raíz cuadrada
raizc( )	1	Devuelve la raíz cúbica de un número
redondear( )	1	Devuelve el número redondeado
rnd( )		
sen( )	1	Devuelve el seno
senh( )	1	Devuelve el seno hiperbólico
tan( )	1	Devuelve la tangente de un ángulo
tanh( )	1	Devuelve la tangente hiperbólico de un número
tau( )	0	Devuelve el valor de TAU
techo( )	1	Devuelve el número redondeado hacia arriba al número entero más cercano
trunc( )	1	Devuelve las partes enteras truncadas de diferentes números

5.47.1 mate.abs( )

El comando **mate.abs( )** devuelve el valor **absoluto (positivo)** del número especificado.

5.47.2 mate.acos( )

Este comando devuelve el **arcocoseno** de un número como un valor entre el **cero (0) y PI radianes**.

Si el número asignado está **fuera** del rango de -1 a 1, el comando devolverá **NaN**.

El valor de **-1** devolverá el valor de PI, y el valor de **1** devolverá el valor de cero (0).

5.47.3 mate.acosh( )

Este comando devuelve el **coseno hiperbólico inverso** de un número.

Este comando admite números igual o mayor que 1 **(x >= 1)**.

Si el número asignado está **fuera** es menor que 1, el comando devolverá **NaN**.

5.47.4 mate.aleatorio( )

El comando **mate.aleatorio( )** devuelve un número.

Este comando también dispone de un alias **mate.alt()**.

Este comando puede admitir desde **cero (0)** parámetros, hasta un máximo de **dos (2)** parámetros.

En este comando se puede asignar números positivos como negativos.

Note: Los parámetros se definen de la siguiente manera:

- **Cero o ningún parámetro:** Devolverá un valor aleatorio entre **cero (0)** y **uno (1)**.
- **Un parámetro:** Se tomará como el número máximo, y devolverá entre **cero (0)** hasta el número asignado.
- **Dos parámetros:** Devolverá un número aleatorio dentro del rango numérico asignado.

5.47.5 mate.asen( )

El comando **mate.asen( )** devuelve el **arcoseno** de un número como un valor entre **-PI/2** y **PI/2** radianes.

Si el número asignado está **fuera** del rango de -1 a 1, el comando devolverá **NaN**.

El valor de **1** devolverá el valor de PI/2 y el valor de **-1** devolverá el valor de -PI/2.

5.47.6 mate.asenh( )

Este comando devuelve el **arcoseno hiperbólico inverso** de un número.

5.47.7 mate.atan( )

Este comando devuelve el **arcotangente** de un número como un valor entre **-PI/2** y **PI/2** radianes.

5.47.8 mate.atanh( )

Este comando devuelve el **arcotangente hiperbólico inverso** de un número.

El parámetro admitido por este comando debe estar entre -0.99 y 0.99.

5.47.9 mate.atan2( )

Este comando devuelve el **arcotangente** del cociente de sus argumentos, como un valor numérico entre **PI** y **-PI** radianes.

El número devuelto representa el ángulo en el sentido contrario de las agujas del reloj en radianes (pero NO en grados) entre los dos valores asignados a la operación.

5.47.10 mate.base( )

Este comando devuelve la **base** de un porciento.

Para el porcentaje **NO es necesario** digitarlo en número decimal sino en **porcentaje**.

Este comando está relacionado con los comandos *mate.parte( )* y *mate.porc( )*

Ejemplo de sintaxis

```
base = parte / porciento
```

5.47.11 mate.cos( )

Este comando devuelve el **coseno** de un número.

El comando **mate.cos( )** devuelve un valor numérico entre **-1** y **1**, que representa el coseno del ángulo.

5.47.12 mate.cosh( )

Este comando devuelve el **coseno hiperbólico** de un número.

5.47.13 mate.e( )

Este comando devuelve el valor del **número de Eular** (*eular's number*).

5.47.14 mate.exp( )

Este comando devuelve el valor E^x, donde E es un número de Euler (aproximadamente 2,7183) y **equis(x)** es el número que se le pasa.

5.47.15 mate.frexp( )

Este comando **descompone** un número en significativo y a una potencia integral de 2.

5.47.16 mate.ldexp( )

Este comando **multiplica** un número por 2, **elevado a una potencia**.

5.47.17 mate.log( )

Este comando devuelve el **logaritmo natural** de un número.

Si el parámetro es **negativo**, devolverá **NaN**.

Si el parámetro es **cero (0)**, devolverá **infinito**.

5.47.18 mate.log10( )

Este comando devuelve el **logaritmo común** de un número (en base a 10).

Si el parámetro es **negativo**, devolverá **NaN**.

Si el parámetro es **cero (0)**, devolverá **infinito**.

5.47.19 mate.parte( )

Este comando devuelve la **parte** de un porciento.

Para el porcentaje **NO es necesario** digitarlo en número decimal sino en **porcentaje**.

Este comando está relacionado con los comandos *mate.base( )* y *mate.porc( )*

Ejemplo de sintaxis

```
parte = base * porciento
```

5.47.20 mate.pi( )

Este comando devuelve el valor de **PI**.

5.47.21 mate.piso( )

A diferencia del comando **mate.techo( )**, el comando **mate.piso( )** redondea un número **hacia abajo** al entero más cercano.

Si el argumento pasado es un número entero, el valor NO se redondeará.

5.47.22 mate.porc( )

Este comando devuelve la **porcentaje** de un porciento.

Este comando también dispone de alias como **mate.porciento( )** y **mate.porcentaje( )**.

Para el porcentaje **NO es necesario** digitarlo en número decimal sino en **porcentaje**.

Este comando está relacionado con los comandos *mate.base( )* y *mate.parte( )*

Ejemplo de sintaxis

```
porcentaje = parte / base
```

5.47.23 mate.pot( )

Este comando devuelve el valor(primer dígito) **elevado a la potencia** (segundo dígito).

5.47.24 mate.max( )

Este comando devuelve el más **alto valor** de una lista.

Este comando también trabaja con números **negativos**.

Este comando NO tiene cantidad máxima de parámetros, lo que significa que se puede hacer una comparación entre 2 a 1000 números si se desea.

5.47.25 mate.min( )

Este comando devuelve el más **bajo valor** de una lista.

Este comando también trabaja con números **negativos**.

Este comando NO tiene cantidad máxima de parámetros, lo que significa que se puede hacer una comparación entre 2 a 1000 números si se desea.

5.47.26 mate.raiz( )

Este comando devuelve la **raíz cuadrada** de un número.

5.47.27 mate.raizc( )

Este comando devuelve la **raíz cúbica** de un número.

5.47.28 mate.redondear( )

Este comando **redondea** el número a su mas cercano entero.

Este comando también dispone de un alias **mate.rnd( )**.

5.47.29 mate.sen( )

Este comando devuelve el **seno** de un número.

El comando **mate.sen( )** devuelve un valor entre **-1** y **1**, que representa el seno del parámetro asignado.

5.47.30 mate.senh( )

Este comando devuelve el **seno hiperbólico** de un número.

5.47.31 mate.tan( )

Este comando devuelve la **tangente** de un número.

5.47.32 mate.tanh( )

Este comando devuelve la **tangente hiperbólica** de un número.

5.47.33 mate.tau( )

Este comando devuelve el valor de **TAU**.

5.47.34 mate.techo( )

A diferencia del comando **mate.piso( )**, el comando **mate.techo( )** redondea un número **hacia arriba** al entero más cercano.

Si el argumento pasado es un número entero, el valor NO se redondeará.

5.47.35 mate.trunc( )

Este comando devuelve la parte **entera truncada** de un número.

Este comando NO redondea el número al más cercado entero, sino **remueve su punto decimal**.

5.48 Lib "sis"

La librería **sis** contiene funciones que nos permitirán operar con nuestro sistema desde Latino.

Lista de comando

Comando	Parámetros	Descripción
dormir( )	1	Detiene el sistema por segundos
ejecutar( )	1	Ejecuta un comando de la consola desde latino
fecha( )	1	Imprime la fecha y hora del sistema (cadena)
salir( )	0	Termina la ejecución de latino
cwd( )	0	Imprime la ruta de donde se está ejecutando Latino
iraxy( )	2	Mueve el cursor de la consola a una nueva posición
tiempo( )	2	Muestra el año, mes, hora, min, y seg de la máquina local
usuario( )	0	Devuelve el nombre del usuario activo del sistema
operativo( )	1	Devuleve el sistema operativo en el que se ejecuta
op( )		

5.48.1 sis.dormir( )

Este comando **detiene la ejecución del código** por la cantidad de segundos signados.

Es importante recalcar que este comando admite **segundos** y NO milisegundos.

5.48.2 sis.ejecutar( )

Con este comando podemos efectuar **comandos nativos de la consola** en el que estamos.

5.48.3 sis.fecha( )

Este comando nos permite obtener la **fecha** del equipo o máquina local en donde Latino se está ejecutado.

Los comando a utilizar son los siguientes:

Comandos	Descripción
seg	Devuelve los segundos
min	Devuelve los minutos
hora	Devuelve las horas (hora militar)
mes	Devuelve el mes
año	Devuelve el año
d_sem	Devuelve el día de la semana
d_mes	Devuelve el día del mes
d_año	Devuelve el día del año
estacion	Devuelve la estación del año

5.48.4 sis.salir( )

Con este comando podemos **cerrar o terminar** la ejecución de Latino.

Este comando es similar al atajo de teclado de consola explicado *aquí*.

5.48.5 sis.cwd( )

Imprime la **ruta** en donde Latino esta siendo ejecutado.

5.48.6 sis.iraxy( )

Con este comando podemos **mover** el cursor de texto a cualquier parte de la ventana.

Este comando es similar al comando **gotoxy( )** en C.

Ejemplo de sintaxis

```
sis.iraxy(valorHorizontal, valorVertical)
```

5.48.7 sis.tiempo( )

A diferencia del comando **sis.fecha( )**, el comando **sis.tiempo( )** nos permite tener un mayor control de las fechas y horas.

Este comando es similar a la librería **datetime** en Python.

Comando	Descripción
%a	Nombre del día de la semana abreviado
%A	Nombre del día de la semana completo
%w	Día de la semana en números del 0-6 (0 es domingo)
%d	Día del mes
%b	Nombre del mes abreviado
%B	Nombre del mes completo
%m	Mes en números
%y	Año abreviado
%Y	Año completo
%H	Horas (00-23)
%I	Horas (00-12)
%p	AM/PM
%M	Minutos (00-59)
%S	Segundos (00-59)
%z	UTC offset
%Z	Zona horaria (timezone)
%j	Número del día del año (001-366)
%U	Números de la semana del año (00-53, Domingo como el primer día de la semana)
%W	Números de la semana del año (00-53, Lunes como el primer día de la semana)
%c	Fecha y el hora de la máquina local
%x	Fecha de la máquina local
%X	Hora de la máquina local
%%	Devuelve el carácter de %

5.48.8 sis.usuario( )

Este comando obtener el **nombre del usuario** activo en el sistema.

5.48.9 sis.operativo( )

El comando **sis.operativo( )** también dispone de una abreviación **op**.

Ambos comandos retornar el nombre del **sistema operativo** es en el que Latino esta siendo ejecutado.

Estés comando solo admite los siguientes comandos (en mayúsculas):

- **WIN32** : para MS-Windows
- **APPLE** : para macOS-X
- **LINUX** : para Linux

5.49 Comandos de Consola

Los comandos de consola son algunas funciones que podemos efectuar con Latino antes de ejecutarlo en nuestra consola o terminal.

A continuación se presentará una tabla con los comandos de consola disponibles en Latino.

Comando	Descripción
-a	Muestra la ayuda de Latino en la terminal
- -ayuda	
- -help	
-e	Ejecuta un comando con código de Latino
-v	Muestra la versión instalada de Latino
- -version	
Ctrl + C	Cierra el programa de Latino en la consola

5.49.1 Comandos

Mostrar menú de ayuda

El menú de ayuda de Latino lo podemos ejecutar con el siguiente comando:

```
latino -a
latino --ayuda
latino --help
```

Ejecutar un comando

Para ejecutar un comando con código de Latino en la consola usamos el siguiente comando:

```
latino -e *código de Latino*
```

Ver versión de Latino

Para mostrar la versión de Latino que se tiene instalada en nuestro sistema, podemos usar el siguiente comando:

```
latino -v
latino --version
```

Cerrar programa de Latino

Si después de iniciar o correr Latino en nuestra consola deseamos salir o cerrar el programa, podemos presionar la siguiente combinación de teclado:

```
Ctrl + C
```

Podemos también salir de Latino escribiendo el comando explicado *aquí*.

5.50 Glosario

Glosario provisional
El contenido de este se irá actualizando periódicamente.

5.50.1 Palabras reservadas

- caso
- cierto | verdadero
- continuar
- defecto | otro
- desde
- elegir
- error
- escribir | imprimir | poner
- falso
- fin
- funcion | fun
- global
- hasta
- imprimirf
- incluir
- mientras
- nulo
- osi
- repetir
- retorno | retornar | ret
- romper
- si
- sino

- tipo

5.50.2 Librerías

- archivo
- cadena
- dic
- gc
- lista
- mate
- paquete
- sis