Guia de bones pràctiques per al desenvolupament de codi en solucions d'Intel·ligència Artificial en salut

© Fundació TIC Salut Social

Aquest informe és fruit de l'Àrea d'Intel·ligència Artificial de la Fundació TIC Salut Social

Autors: Susanna Aussó, Didier Domínguez, Yeray Bartolomé

Edició electrònica: desembre de 2022

Aquesta obra està subjecta a una llicència de Reconeixement - No Comercial - Sense Obres Derivades 4.0 de Creative Commons. Se'n permet la reproducció, distribució i comunicació pública sempre que es reconeguin l'autoria i l'editor i no se'n faci un ús comercial. No és permesa la transformació d'aquesta obra per generar una nova obra derivada.

/Índex de contingut

01/ Introducció 04

02/ Llenguatges de programació en IA 06

2.1. Python 08
2.2. R 08
2.3. Java 09
2.4. C++ 09
2.5. Rendiment i empremta ecològica dels llenguatges de programació 10

03/ Metodologies de desenvolupament del programari 11

04/ Recomanacions pel desenvolupament de codi 14

4.1. Nomenclatura 16
4.2. Ordre 17
4.3. Estil 18
4.4. Rendibilitat 20
4.5. Documentació 21

05/ Estàndards de codi 22

5.1. Estandards de programació en R 23
5.2. Estàndards de programació en Python 24
5.3. Estàndards de programació en C, C++, i altres 24
5.4. Beneficis dels estàndards de codi en el desenvolupament de programari 25
5.5. Principis SOLID 27

06/ Qualitat del codi 28

6.1. Anàlisi estàtica de qualitat del codi 29
6.2. Mètriques de qualitat de codi 30
6.2.1.Complexitat ciclomàtica 30
6.2.2.Complexitat de Halstead 31
6.3. Proves 32
6.3.1. Proves unitàries 33
6.3.2. Proves del sistema 33
6.3.3. Proves d'implantació 34
6.3.4. Proves d'acceptació 34
6.3.5. Proves de regressió 34

07/ Certificacions ISO 35

08/ Referències 38

09/ Annexos 40

ANNEX 1: Tidyverse Style Guide 41
ANNEX 2: Guia de bones pràctiques de programació en R de Google 44
ANNEX 3: Guia de bones practiques de programació en R d'Amazon AWS 47
ANNEX 4: Guia de bones pràctiques per Python – PEP 8 49
ANNEX 5: Guia de bones pràctiques per Python de Google 53
ANNEX 6: Regles i directrius de MISRA C:2012 56

1. Introducció

El Programa per a la promoció i desenvolupament de la Intel·ligència Artificial al Sistema de Salut de Catalunya té com a finalitat crear un entorn que faciliti el desenvolupament i la implementació de solucions d'Intel·ligència Artificial (IA) per a l'optimització de processos al sistema sanitari català.

La Fundació TIC Salut Social ha creat aquesta guia amb l'objectiu de donar suport a les parts involucrades en el desenvolupament de codi d'algorismes d'IA aplicats a l'àmbit de la salut.

El seguiment d'aquest conjunt de bones pràctiques permet l'obtenció d'un codi més comprensible, reutilitzable, escalable i més fàcil de modificar, que ajudarà a disminuir el temps de dedicació en les tasques de revisió i a prevenir errors comuns durant les diferents etapes del projecte. Aquestes recomanacions també faciliten el treball de forma col·laborativa, ja que permeten que la resta de col·laboradors i col·laboradores que també desenvolupen codi puguin fer una comprensió àgil i, si és necessari, desenvolupar nous avenços sobre el projecte seguint una metodologia específica.

Una part fonamental per mantenir unes bones pràctiques durant el desenvolupament de programari són els estàndards de codi. Entenem com a estàndard de codi un conjunt de normes i recomanacions establertes (denominacions, formats, etc.) per a l'escriptura de codi que varien depenent del llenguatge de programació triat. Aquests estàndards han anat sorgint de les diferents comunitats de desenvolupadors i desenvolupadores amb el pas del temps i, si s'apliquen adequadament, poden incrementar notablement la qualitat i la gestió del codi.

La IA abraça un ampli ventall de tècniques, mètodes i aproximacions que es poden implementar en múltiples llenguatges de programació. Donat l'augment de la complexitat de les aplicacions d'IA en l'àmbit de la salut i la seva ràpida expansió en els darrers temps, es fa necessària l'adopció d'estàndards de codi coherents en el desenvolupament de programari.

Actualment no hi ha cap conjunt general dels requisits que ha de complir el programari per als dispositius mèdics, ja sigui d'organitzacions internacionals de normalització o d'organismes reguladors mèdics clau com l'Agència Europea de Medicaments i l'Administració d'Aliments i Medicaments dels EUA. Aquests darrers organismes requereixen que els desenvolupadors i desenvolupadores de dispositius validin adequadament la seguretat i la ciberseguretat de les seves tecnologies, però hi ha pocs detalls sobre com s'ha d'aconseguir la seguretat dels dispositius. Per tant, tot i que no hi ha unes directrius clares que obliguen directament a utilitzar uns estàndards determinats de codi, és evident que, a mesura que les bases de codi es fan més complexes i connectades, els estàndards de codi es tornen més rellevants per als equips de programadors.

2.
Llenguatges de programació en IA

Existeixen nombrosos llenguatges de programació capaços de satisfer diferents necessitats en el disseny i desenvolupament de diferents programaris destinats a la IA, ja sigui per emmagatzematge i consulta de dades, anàlisi de les mateixes, entrenament de models o tècniques d'explicabilitat, i sovint aquests llenguatges aconsegueixen comprendre un ventall d'aquestes funcionalitats simultàniament.

L'índex TIOBE [1] és un indicador de la popularitat dels llenguatges de programació creat per aquesta empresa amb la participació de la comunitat de persones programadores. L'índex s'actualitza un cop al mes. Les qualificacions es basen en el nombre d'enginyers i enginyeres qualificats a tot el món, cursos i proveïdors de tercers.

Oct 2022	Oct 2021	Change	Programming Language	Ratings	Change
1	1		Python	17.08%	+5.81%
2	2		C	15.21%	+4.05%
3	3		Java	12.84%	+2.38%
4	4		C++	9.92%	+2.42%
5	5		C#	4.42%	-0.84%
6	6		Visual Basic	3.95%	-1.29%
7	7		JavaScript	2.74%	+0.55%
8	10	↑	Assembly language	2.39%	+0.33%
9	9		PHP	2.04%	-0.06%
10	8	↓	SQL	1.78%	-0.39%
11	12	↑	Go	1.27%	-0.01%
12	14	↑	R	1.22%	+0.03%

Figura 1. ÍNDEX TIOBE

Per tal de simplificar la tasca d'escollir els llenguatges a emprar, en aquesta guia es fa focus en Python, Java, C++ i R, donat que són dels llenguatges de mig i alt nivell més emprats de la indústria de producció de software [Fig.2, 2], i R és un dels llenguatges dedicat al tractament i visualització de dades més emprats a la recerca científica. De tots aquests es disposa d'una gran quantitat de documentació i projectes per tal de facilitar la tasca de seleccionar el més adequat a les necessitats del software. Tot i així, s'hi val saber que existeixen altres llenguatges bastant emprats en IA com són Julia, Haskell, Lisp, JavaScript, Prolog i Scala, que són llenguatges més moderns amb una quantitat significativament inferior de documentació en aquest camp, tot i que és recomanable contemplar-los.

2.1 Python

Python és un llenguatge de programació de 4a generació, d'alt nivell i interpretat [2]. Això vol dir que no requereix d'ésser compilat prèviament sinó que es pot executar directament amb un intèrpret. Això implica l'avantatge de ser més independent del maquinari i entorn d'execució, o que sigui possible executar el codi en remot en temps real, com n'és l'exemple de Google Colab. Respecte a les variables, és un llenguatge sense tipat i amb declaració de variables discreta, ja que només cal assignar un valor a una variable prèviament inexistent per tal d'inicialitzar-la. L'intèrpret automàticament assigna el tipus en temps d'execució en funció dels valors assignats. Python permet carregar i processar dades, visualitzar-les, generar estadístics bàsics i complexos, desenvolupar xarxes neuronals, algorismes genètics, etc. A banda de la seva popularitat per la seva senzillesa i versatilitat, el principal motiu per utilitzar Python per desenvolupar IA és l'ecosistema de recursos disponibles que disposa. Per exemple, els paquets i eines ja existents d'aprenentatge automàtic i aprenentatge profund més coneguts i utilitzats com Pandas, NumPy, TensorFlow, Keras, Scikit-learn i molts més, estan tots desenvolupats amb Python.

El llenguatge ha evolucionat en paral·lel a la disciplina d'IA, i això fa que sigui gairebé indispensable conèixer-lo si es treballa en aquest camp. A més, proporciona un codi molt clar, permet un prototipat ràpid, i facilita la col·laboració per treballar en equip.

2.2 R

R , al igual que Python, és un llenguatge de programació de 4a generació, alt nivell i interpretat [3]. També comparteixen el tipat i declaració discretes de variables. Si bé s'ha relacionat tradicionalment amb entorns més acadèmics, es tracta d'un llenguatge específic per treballar en anàlisi de dades i aprenentatge automàtic.

És un llenguatge de codi obert, multiplataforma, funcional, orientat a objectes i fàcil d'aprendre. Disposa d´un entorn de desenvolupament molt popular per al llenguatge anomenat RStudio (el qual ara també suporta Python).

2.3 Java

Java[1] és un llenguatge de 3a generació d'alt nivell d'interpretació mixta, doncs es precompila en binari però s'executa per un intèrpret anomenat Java Virtual Machine (JVM) [4]. Java és un llenguatge amb tipat fort estàtic i declaració explícita de variables. Pensat per crear scripts, macros i material similar, el seu principal avantatge és la seva senzillesa d'aprenentatge per a les coses comunes i que funciona tant en navegadors com en servidors (a través de plataformes com Node.js).

És un gran complement d'altres llenguatges de programació com Python o R, especialment per a la visualització. És un llenguatge multi-paradigma que segueix els principis orientats a objectes i al principi Write Once/Run Anywhere (WORA). Aquest principi significa que Java és un llenguatge que pot ser executat en qualsevol plataforma que el suporti sense necessitat de compilació, i no només és adequat per a algorismes de cerca o de processament de llenguatge natural sinó també per a xarxes neuronals. Això és possible gràcies a la JVM (Java Virtual Machine), una mena d'emulador que simula una arquitectura comuna dintre de cada SO per la qual està distribuïda, que permet un desacoblament entre aquest SO i l'aplicació que es pretén executar.

2.4 C++

C++ és un llenguatge de 3a generació precompilat [4]. Es caracteritza per ser de mig i alt nivell, doncs normalment s'executa sobre el sistema operatiu, però ofereix des de funcions de baix nivell heretades del llenguatge C, fins a programació orientada a objectes i programació web. Entre els esmenats, C++ és el llenguatge de programació més ràpid, i la seva velocitat s'aprecia per a projectes de programació d'IA que són sensibles al temps. Proporciona una execució ràpida i amb poc temps de resposta, i s'aplica a cercadors i al desenvolupament de jocs d'ordinador. Això és degut a que C++ és una evolució de C i està escrit en el mateix, i per tant està dissenyat per tal d'oferir implementacions a baix nivell que permeten aprofitar al màxim el maquinari del qual es disposa a canvi d'un major acoblament a l'arquitectura en què es suporta el codi. A més, C++ permet un ús extensiu d'algorismes i és eficient en l'ús de tècniques estadístiques d'IA. Un altre factor important és que C++ admet la reutilització de programes en desenvolupament a causa de l'herència i l'ocultació de dades, i per tant és eficient en temps i estalvi de costos.

Les seves extenses llibreries són ideals per a codis complexos d'IA, classificació, càlculs matemàtics més ràpids i aplicacions d'alt rendiment.

[1] https://www.java.com/es/

2.5

Rendiment i empremta ecològica dels llenguatges de programació

En la següent taula es poden veure de forma sintetitzada el consum en Watts (w) dels llenguatges exposats, basats en diferents quantitats d'inputs per mesurament:

ALGORISME	PYTHON	R	C++	JAVA
Heap Sort (100 inputs)	5.4	6	17.3	5.2
Heap Sort (1000 inputs)	8.1	9.4	17.6	8.2
Heap Sort (1500 inputs)	13.4	12.8	18	14.1
Quick Sort (100 inputs)	6.8	8.1	17.2	6.2
Quick Sort (1000 inputs)	14.4	15.3	17.8	14.2
Quick Sort (1500 inputs)	16.3	16.6	20.2	17
Selection Sort (100 inputs)	6.1	6.4	17.4	5.8
Selection Sort(1000 inputs)	6.7	7.1	18	6.9
Selection Sort(1500 inputs)	8	8.2	20.2	8.3
Matrix Addition (2x2)	0.8	1.1	17.7	0.7
Matrix Addition (5x5)	2.2	2.3	18.9	2.4
Matrix Addition (10x10)	3.1	3.2	19.6	3.2
Matrix Multiplication (2x2)	0.9	1	17.5	0.7
Matrix Multiplication (5x5)	1.6	1.8	18.2	1.6
Matrix Multiplication (10x10)	2.8	3	18.6	2.9

Taula 1. Consum energètic de diferents algorismes sobre els llenguatges esmenats.

Com es pot observar, dels llenguatges tractats, C++ n'és el que implica un major consum energètic de base, però és el que menys creix proporcionalment al volum d'inputs proporcionats. Mentrestant, Java, Python i R, al ésser interpretats, consumeixen un volum de Watts molt més reduït quan es tracta d'operacions simples, però que escala linealment amb el número d'inputs. En qualsevol cas, Python està considerat el llenguatge més ràpid i eficient energèticament per a la majoria de tasques de codi.

Avui dia, la majoria de projectes de software han de suportar una naturalesa de desenvolupament dinàmica i iterativa, al mateix temps que han de suposar una eina factible i profitosa per a un projecte canviant, escalable i migrable. Per aquests motius, des de la pròpia indústria, treballs de personalitats com Hirotaka Takeuchi i Ikujiro Nonaka [5] i Jeff Sutherland [6, 7] i Ken Schwaber [8] van modelar les filosofies de desenvolupament de software anomenades Agile (o àgils en Català). El paradigma Agile, que es centra en cicles curts de desenvolupament on totes les activitats del mateix s'executen de manera simultània (descobriment de requeriments, implementació, testeig, incloent revisió del treball de cicles anteriors), és una filosofia "proposada per sobreposar-se a les limitacions del mètode de desenvolupament convolucional" [9], és a dir, pretén minimitzar l'impacte del desenvolupament en cascada, atorgant certa flexibilitat a l'hora d'adoptar canvis en els requeriments en cada fase del desenvolupament. Aquesta manera de treballar és una bona pràctica que minimitza els errors posteriors del programari en producció.

Dintre de la filosofia Agile, i al llarg dels anys, s'han elaborat i establert diverses metodologies de desenvolupament, cadascuna amb els seus avantatges i punts febles, amb l'objectiu d'adaptar aquesta filosofia a la naturalesa de diferents projectes de software, equips de desenvolupament, i polítiques d'organitzacions. Dintre d'aquestes, en destaquen tres:

- **Metodologia Scrum:** enfocada en ser un marc de treball que divideix el desenvolupament del projecte en diversos sprints de temps fixe. Dintre de cada sprint, s'han d'efectuar diferents tasques referents al desenvolupament de les funcionalitats planificades per aquest, com són la revisió de requeriments propers, la implementació de requeriments presents i integració i/o refactorització dels anteriors, el testeig de integral del producte abans de llançar la nova versió, i la redacció de documentació, però fent èmfasi en el codi com a principal font de la mateixa.

- **Metodologia Kanban:** consisteix en reflectir les tasques de cada iteració en etiquetes que es poden moure en columnes segons el seu estat de desenvolupament, i està enfocada a mostrar el progrés del producte a persones externes a l'equip de desenvolupament.

- **Metodologia Lean:**
  - Eliminar desaprofitaments/restes: per desaprofitaments o restes considerem tot allò que no aporta valor al programari. Dins del desenvolupament de programari podríem incloure els següents elements: codi generat que ofereix funcionalitats no desitjades o necessàries, retards en el procés de desenvolupament de programari, problemes amb la comunicació interna, documentació excessiva, etc. No obstant això, a vegades en temes d'IA es treballa amb llibreries i algorismes no deterministes o d'una complexitat elevada i hem de tenir en compte un parell de conceptes i bones pràctiques. Quan la depuració és difícil, sovint és un signe que el codi no té alguna funcionalitat de diagnòstic o traçabilitat. Aquest aspecte de l'escriptura de codi sovint es passa per alt perquè no és funcional, però pot estalviar hores i fins i tot dies de vegades de temps de depuració.
  - Potenciar l'equip: ajudar a que les persones desenvolupadores participin en la presa de decisions de temps associats a les tasques, priorització de les mateixes i altres formes d'implicar a l'equip, faciliten que aquestes persones se sentin part important en ell. A més, els propis desenvolupadors i desenvolupadores saben de primera mà quines tasques costen més, quines menys i quines implicacions tenen en el cicle de vida del projecte. Tècniques en les quals s'assignen pesos, esforços o complexitats a les tasques a realitzar cauen dins d'aquest principi.
  - Integració contínua: disposar d'un bon sistema d'integració contínua que inclogui proves automatitzades, compilacions i proves d'usabilitat és crític, ja que ens permetrà produir un programari fàcil de mantenir, de millorar i de reutilitzar. Amb això evitarem afegir restes a aquest programari i facilitarà el manteniment del codi i de posteriors versions en producció.
  - Visualitzar tot el conjunt: analitzar les interaccions del nostre programari amb la resta de sistemes ens permetrà estudiar possibles millores i canvis que redundin en un millor rendiment i aportin un major valor a les persones usuàries finals i l'equip del projecte.

4. Recomanacions per al desenvolupament de codi

Definim cinc conceptes principals en l'aplicació de les bones pràctiques per programar: la nomenclatura, l'ordre, la rendibilitat, l'estil i documentació. Cadascun d'ells engloba diferents apartats:

- **Nomenclatura**
  - Convencions de denominació de variables
  - Convencions de nomenclatura de classes i funcions
- **Ordre**
  - Afegir comentaris clars i concisos al codi
  - Agrupació i organització del codi
- **Estil**
  - Limitar la longitud de la línia
  - Evitar utilitzar "números màgics"
  - Sagnats
  - Evitar la nidificació profunda
- **Rendibilitat**
  - Portabilitat
  - Reutilització i escalabilitat
  - Test per verificar la funcionalitat
- **Documentació**
  - Documentació i/o arxiu README amb explicació i guia utilitzada pel codi

A continuació s'expliquen detalladament aquests conceptes, que configuren l'estructura general de bones pràctiques.

4.1. Nomenclatura

Convencions de denominació de variables

Durant el moment de programar, els noms de les variables han de ser fàcil d'entendre i de representar les dades que emmagatzemen. Per tant, la manera com s'anomenen les variables és clau per fer que un codi sigui llegible i evitar confusions.

La idea de nomenar variables durant el desenvolupament de codi és senzilla: crear noms de variables que s'expliquin per sí mateixos i que segueixin una coherència al llarg del codi. Intentar estalviar temps utilitzant noms molt curts per a variables i funcions és contraproduent a la llarga, sobretot quan hi han molts editors de codi i IDE[2] disponibles per ajudar a escriure codi.

Els noms han de tenir límits de paraula. Hi ha tres opcions populars:

- **UpperCamelCase o PascalCase:** la primera lletra de cada paraula s'escriu en majúscula (per exemple ParseRawImageData()).
- **camelCase:** la primera lletra de cada paraula s'escriu en majúscula, excepte la primera paraula (per exemple parseRawImageData()).
- **Guions_baixos:** guions baixos entre paraules, com mysql_real_escape_string().

Algunes plataformes solen utilitzar un determinat esquema de denominació. En el cas de Java i C++, la convenció més emprada és utilitzar camelCase per variables i mètodes, mentre que PascalCase s'empra per constructors i noms de classe. En el cas de Python i R, s'acostuma a emprar guions baixos per als noms de variables.

Aquests estils també es poden barrejar. Alguns programadors i programadores prefereixen utilitzar guions baixos per a funcions de procediment i noms de classe, però utilitzen CamelCase per als noms de mètodes de classe.

[2] Un ambient de desenvolupament integrat (Integrated Development Environment), a diferència d'un editor, és un programa més pesat que demana molta més memòria RAM i un processador més potent.

Convencions de nomenclatura de classes i funcions

Els conceptes bàsics de la denominació de classes i funcions són un aspecte essencial per aprendre a programar.

De manera similar a les convencions de denominació de variables, les funcions i les classes també haurien de consistir en títols descriptius que es delimiten mitjançant l'ús de convencions, com s'ha esmentat anteriorment. El propòsit d'utilitzar les convencions de denominació adequades és assegurar-se que les variables, funcions i classes del codi es poden distingir fàcilment entre si.

4.2. Ordre

Afegir comentaris clars i concisos al codi

Els codis sempre s'acaben modificant o actualitzant amb el pas del temps, i gairebé totes les persones programadores es trobaran amb el codi d'una altra persona en un moment o altre. Un mal hàbit entre els programadors i programadores sense experiència és incloure pocs comentaris durant el desenvolupament de programari, fet que suposa un repte important quan es treballa en equip, on més d'una persona pot estar treballant en un mòdul concret.

Per altra banda, és recomanable no exagerar. L'excés de comentaris pot disminuir el valor de la transferència de coneixement entre les persones desenvolupadores que treballen en el mateix mòdul. Idealment, els comentaris haurien d'explicar per què s'està utilitzant cert codi. Si s'ha d'escriure un comentari de més d'una línia per explicar què està fent el codi, caldria considerar reescriure el codi perquè sigui més llegible.

En resum, existeix un gran debat entre si és un bon estàndard comentar codi o no. Avui dia, una part important del sector tecnològic considera que el bon codi és aquell que, per emprar una nomenclatura, estructura i format adequats, és llegible sense la necessitat de comentaris. En qualsevol cas, val la pena explicar que els comentaris no són la forma correcta de transmetre informació sobre la funcionalitat i estructura del projecte.

Agrupació i organització del codi

Una bona organització és vital per mantenir una bona estructura i que el codi s'entengui. Molt sovint, certes tasques requereixen vàries línies de codi. Afegir un comentari al principi de cada bloc de codi emfatitza la separació visual i ajuda a la comprensió. Mantenir tasques similars dins de blocs de codi separats, amb alguns espais entre ells, tal com es mostra en el següent exemple, és molt positiu per a la bona gestió del codi:

```
# llibreries
import argparse
import os
import sys
import numpy as np
import pandas as pd

# moduls
# carregar llistat de gens
def load_genes(genes):
    list_genes = open(str(genes), "r")
    genes = [ ]
    for line in list_genes:
        line = line.strip()
        genes.append(str(line))
    return genes
```

Figura 2. Exemple d'agrupació i organització del codi.

4.3.

Estil

Un estil de programació és un conjunt de directrius que s'utilitzen per donar format a instruccions de programació. Seguir un estil facilita a les persones programadores la comprensió del codi, el manteniment i ajuda a reduir la probabilitat d'introduir errors. Les directrius es poden desenvolupar a partir de les convencions de codificació utilitzades en una organització amb variacions d'estil per a diferents llenguatges de programació.

Els elements clau de la guia d'estil de programació inclouen les convencions de denominació, l'ús de comentaris i el format (sagnia, espais en blanc, etc.). En alguns llenguatges (per exemple, Python), el sagnat s'utilitza per indicar estructures de control (per tant, es requereix un sagnat correcte), mentre que en altres llenguatges el sagnat s'utilitza per millorar l'aparença visible i la llegibilitat del codi (per exemple, Java).

Limitar de la longitud de la línia

Una bona pràctica és evitar escriure línies de codi massa llargues horitzontalment, ja que els nostres ulls se senten més còmodes quan llegeixen columnes de text altes i estretes. Per exemple, quan s'utilitza el llenguatge Python es recomana que el límit de llargada de les línies sigui de 79 caràcters.

```
#unload alternatives
refbases = set(refBase.split (',’))
altBases = set(altBase. split (',’))
added = set ()
for refBase in refBases.difference(altBases):
        for altBase in filter (lambda x: x != refBase, altBases):
                outline = process_alt(refBase, altBase, contig, pos,

                out.write(outline)
                added.add((refBase, altbase))
for altBase in altBases.difference(refBases):
        for refBase in filter(lambda x: x != altBase, refBases) :
                if (refBase, altBase) in added:
                        continue
                outline = process_alt(refBase, altase, contig,
```

Figura 3. Exemple limitació de la longitud de la línia.

Evitar utilitzar "números màgics"

El concepte de "números màgics" a la programació es refereix a l'ús de valors numèrics codificats en el codi. L'ús d'aquests números pot tenir sentit per a qui escriu el codi, però pot dificultar entendre què feia aquest número quan en un futur la persona programadora es miri el mateix fragment de codi o ho faci una altra persona.

```
#cas 1
thr = 400
for gene in results:
  p = results[gene]
    if p <= thr:
      print gene, p

#cas 2
for gene in results:
  p = results[gene]
    if o <= 400:
      print gene, p
```

Figura 4. Exemple de com no utilitzar números màgics.

Observant el codi de la figura 4, en el primer cas es veu clarament que s'està comprovant si el recompte és inferior al llindar seleccionat (thr = 400), mentre que en el segon cas es desconeix el significat del número 400. Addicionalment, és més fàcil actualitzar el llindar thr canviant-ne el valor en un sol lloc, ja que això no seria possible utilitzant números màgics, perquè caldria revisar tot el codi per entendre d'on ve el valor.

Sagnats

El format i el sagnat són necessaris per organitzar el codi. Mitjançant l'ús de sagnats, espais en blanc i pestanyes dins del codi, les persones programadores asseguren que el seu codi sigui llegible i organitzat.

No hi ha cap manera correcta o incorrecta de sagnar un codi, hi ha opinions populars però no se segueix cap patró de manera universal. L'important és ser coherent amb l'estil escollit; que els estils de sagnat canviïn a la meitat d'un script porta a problemes de confusió i errors en l'execució de codi. En segons quins llenguatges (per exemple Python) no es permet la combinació de diferents estils de sagnat (tabulador i espais). Si formeu part d'un equip o si esteu aportant codi a un projecte, hauríeu de seguir l'estil existent que s'està utilitzant en aquest projecte.

Els estils de sagnat no sempre són completament diferents els uns dels altres. De vegades, barregen regles diferents. Per exemple, en llenguatge R, la clau d'obertura "{" d'una funció va a la mateixa línia que les estructures de control, però després la definició de les funcions van a la línia següent i en una posició anterior, com es veu en la següent imatge:

```
y <- 10
y <- function(x) {
  x+y
}
f (5)
```

Figura 5. Sagnat i utilització de claudàtors.

Evitar la nidificació profunda

Massa nivells d'imbricació poden dificultar la lectura i el seguiment del codi. Per facilitar la lectura, normalment és possible fer canvis al codi per reduir el nivell d'imbricació:

```
#cas 1
thr1 = 400
thr2 = 1000
thr3 = 100
for gene in results:
  p = results[gene]
  if p <= thr1:
    print gene
      if p <= thr2:
        print gene
          if p <= thr3:
            print gene
    ...
```

```
#cas 2
thr1 = 400
thr2 = 1000
thr3 = 100
for gene in results:
  p = results[gene]
  if p <= thr1:
    print gene
  elif p <= thr2:
    print gene
  elif p <= thr3:
    print gene
  else:
    ...
```

Figura 6. Exemple de bona pràctica en nidificació en codi.

4.4. Rendibilitat

Portabilitat

La portabilitat és la capacitat del codi font d'executar-se en diferents màquines i plataformes tant com sigui possible. Si quan el programari es transfereix d'un entorn a un altre les persones programadores han de tornar a escriure el mateix codi, suposa una pèrdua de temps i esforç. El suport de múltiples plataformes es pot planificar al començament del desenvolupament del programari; es pot escriure codi que podria funcionar en tots els entorns possibles.

La portabilitat és un aspecte clau que garanteix la funcionalitat del programa. Si el codi conté valors específics al codi font (hard-code) de paràmetres ambientals, com ara noms d'usuari, noms d'amfitrió, adreces IP o URL, no s'executarà en un host amb una configuració diferent. En aquestes situacions, la persona programadora hauria de canviar el codi font i no ajudaria a la portabilitat. Per fer front a això, caldria "parametritzar" variables i configurar-les abans d'executar el programari en diferents entorns. Això es pot controlar amb fitxers de propietats o configuració, bases de dades o servidors d'aplicacions.

A més, els recursos com els fitxers XML també han de tenir variables en lloc de valors literals. En cas contrari, caldria canviar les referències mentre es codifica cada vegada que es vulgui portar l'aplicació a un altre entorn.

Una altra bona pràctica és la creació amb Docker de contenidors d'aplicacions portables i autosuficients. Els contenidors permeten crear una aplicació amb totes les biblioteques i dependències que necessita i distribuir-la com un sol paquet. L'aplicació s'executarà en qualsevol altra màquina de l'entorn de producció, independentment de les configuracions que la màquina pugui tenir i que puguin diferir de la màquina utilitzada per escriure i el codi.

Reutilització i escalabilitat

En el desenvolupament de codi, la reutilització és un objectiu essencial del disseny de programari. Si els mòduls i components ja s'han provat, es pot estalviar temps reutilitzant-los. Els projectes de programari sovint comencen amb un marc o estructura existent que conté una versió anterior del projecte. Per tant, reutilitzant components i mòduls de programari es pot reduir el temps, els costos i els recursos de desenvolupament. I això, es tradueix directament en un lliurament més ràpid del projecte, augmentant així la rendibilitat.

Un altre aspecte clau al qual cal prestar atenció és l'escalabilitat del codi. A mesura que canvien les demandes de les persones usuàries, s'afegeixen constantment noves funcions i millores a una aplicació. Per tant, la capacitat d'incorporar actualitzacions és una part essencial del procés de disseny del programari. En aquest sentit, una bona pràctica consisteix en testejar el codi dels algoritmes d'IA amb subconjunts petits de dades i una vegada tenim la certesa de que no hi ha errors, podem escalar amb dades d'una grandària major.

Test per verificar la funcionalitat

Provar el treball mentre es codifica és una part vital del desenvolupament de programari i s'ha de planificar bé, ja que requereix que els casos de prova estiguin preparats abans que comenci el desenvolupament del programari. A més, tot i que les proves d'unitats bàsiques són una bona pràctica per adoptar, també és útil realitzar proves funcionals automatitzades amb l'ajuda d'eines com Unittest, Doctest, Geb, Spock o Selenium.

Un cop el programari ha finalitzat i es posa a disposició de la persona usuària final, depenent de l'objectiu del programari realitzat, es recomana l'opció de posar un executable de prova per quan la persona usuària instal·li el programari en el seu entorn local. D'aquesta manera és fàcil testejar si l'entorn de la persona usuària és adequat o no abans que aquest es posi a realitzar tasques més complexes.

4.5. Documentació

Una bona documentació de tot el que s'ha dut a terme és essencial per a la correcta execució del programari desenvolupat. Els documents bàsics per tal de tenir un projecte ben ordenat, executable i de fàcil accés per obtenir versions futures són un README, INSTALL i LICENSE. En aquests documents hi ha escrita l'estructura del codi, les guies que s'han utilitzat per programar-ho, els passos a seguir per una correcta instal·lació i l'especificitat de la llicència sota la qual el programari es posa a disposició de l'usuari o usuària públic.

5. Estàndards de codi

Un estàndard de codi és un paradigma de programació que busca reduir el nombre de decisions que la persona programadora ha de prendre en el moment d'escriure el codi. Companyies com Google, on Python és el seu principal llenguatge de programació, poden crear els seus propis estàndards de codi per garantir la coherència de les convencions de denominació i la disposició del codi. També, llibreries de programari lliure (com per exemple Tensorflow[3]), poden emprar estàndards prèviament establerts i adaptar-los a les seves necessitats. Sovint, aquestes guies acaben sent un referent per als programadors i programadores i s'han de tenir en compte.

A continuació s'exposen els estàndards més reconeguts i utilitzats per programació en R, Python, C, C++ i altres, que fan que el codi sigui més fàcil de llegir, compartir i verificar.

5.1. Estandards de programació en R

Tidyverse Style Guide

Tidyverse Style Guide[4] de Hadley Wickham consisteix en un seguit de normes i recomanacions per tal de programar R de manera segura i correcta. Es basa en dos grans apartats: anàlisi i creació de llibreries. L'apartat d'anàlisi se centra en la nomenclatura dels noms, la sintaxi, la creació i estructura de funcions i el correcte funcionament de les pipes (procés de passar d'una funció a un altre). Pel que fa a creació i distribució de llibreries, també se centra en gran part en la nomenclatura del nom i organització dels fitxers, en la bona documentació i descripció de les funcions, en la creació d'un fitxer test per la validació de codi i la correcta comunicació d'errors en la funcionalitat del codi (veure Annex 1).

Google R Style Guide

La Google R Style Guide[5] és una bifurcació de la guia Tidyverse Style Guide explicada anteriorment. Les modificacions de Google es van desenvolupar en col·laboració amb la comunitat interna de persones usuàries de R i reafirmen les normes i recomanacions de Hadley Wickham per tal de programar R de manera eficient i correcta. Les dues principals diferències són la identificació de funcions amb BigCamelCase per distingir clarament de la resta d'objectes, i la utilització explícita de l'opció return(), entre altres (veure Annex 2).

R Coding Style (Amazon)

La guia de programació en R d'Amazon (R Coding Style[6]) es basa en la guia Tidyverse Style Guide, la guia d'estil de R de Google (ambdues explicades prèviament) i en la guia "Advanced R" de Hadley Wickham [10]. Donada l'elevada generació de contingut en la seva plataforma, Amazon va decidir crear la seva pròpia guia on es recullen un seguit de recomanacions sobre llenguat-

[3] https://www.tensorflow.org/community/contribute/code_style
[4] https://style.tidyverse.org/index.html
[5] https://google.github.io/styleguide/Rguide.html
[6] https://rstudio-pubs-static.s3.amazonaws.com/390511_286f47c578694d3dbd35b6a71f3af4d6.html

ge, estil i el bon ús i que està en constant actualització. Aquesta guia conté aspectes de nomenclatura, de sintaxi i d'organització (veure Annex 3).

5.2. Estàndards de programació en Python

Python Enhancement Proposals: PEP8

El principal benefici de la programació en Python és la fàcil lectura del llenguatge. La guia PEP8[7], escrita per Guido van Rossum, Barry Warsaw i Nick Coghlan, ofereix un seguit de regles i recomanacions per a la programació en Python per tal de mantenir la fàcil lectura i escriptura. Aquest conjunt de recomanacions ha anat evolucionant i actualitzant-se amb el temps, i s'ha anat adaptant amb l'evolució del llenguatge de Python. La guia conté conceptes que van des de la sintaxi fins a l'organització estructural del codi (veure Annex 4).

Google Python Style Guide

La Google Python Style Guide[8] conté totes les normes i recomanacions per programar en Python. Cada projecte de codi obert té la seva pròpia guia de codi i bones pràctiques. Han generat la seva pròpia guia on es recullen un seguit de recomanacions sobre llenguatge, estil i bon ús (veure Annex 5).

The Hitchhiker's Guide to Python

La guia Hitchhiker de Python[9] és una guia no oficial pública i actualitzada constantment per persones programadores de Python que permet tenir un manual de bones pràctiques amb les recomanacions per a una correcta instal·lació, configuració i ús de Python [11].

5.3. Estàndards de programació en C, C++, i altres

MISRA C/C++

MISRA[10] ofereix directrius de bones pràctiques líders a nivell mundial amb la finalitat de proveir portabilitat, seguretat i fiabilitat al codi font en el context de programari integrat. Va començar a principis de la dècada de 1990 com un projecte del programa *SafeIT* del govern del Regne Unit, i va desenvolupar directrius per a la creació de programari incrustat en sistemes electrònics de vehicles de carretera. Amb el temps, MISRA ha continuat treballant de manera voluntària, produint publicacions de referència com ara MISRA C i MISRA C++ [12, 13].

Actualment, s'utilitza el MISRA C:2012, publicat al 2013 i actualitzat l'any 2020, el qual consta de 17 directrius – que engloben compliments generals no relacionats amb el codi font (requisits, especificacions, disseny, ...) i 158 regles – que engloben informació relacionada amb el codi font (veure Annex 6).

CERT

Un Equip de Resposta a Emergències Informàtiques (en anglès Computer Emergency Response Team, CERT[11]) és un grup d'experts en seguretat de la informació responsable de la protecció, detecció i resposta als incidents de ciberseguretat d'una organització.

[7] https://peps.python.org/pep-0008/
[8] https://google.github.io/styleguide/pyguide.html
[9] https://docs.python-guide.org/
[10] https://www.misra.org.uk/
[11] https://wiki.sei.cmu.edu/confluence/

Els estàndards de codi CERT/CC consisteixen en regles i recomanacions que permeten a les persones desenvolupadores evitar pràctiques de codi insegures i comportaments no definits que poden provocar vulnerabilitats. Aquests estan en constant revisió, amb les últimes normes i recomanacions disponibles en línia[12].

L'entitat ha desenvolupat quatre subconjunts de regles i recomanacions per donar suport a quatre llenguatges de programació: CERT C, el qual està centrat en ajudar a reduir la probabilitat de vulnerabilitats de llenguatge C; CERT C++, on moltes de les normes i recomanacions són compartides amb CERT C però n'afegeix d'específiques relacionades amb C++ per tal d'evitar vulnerabilitats específiques d'aquest; CERT Oracle per Java, el qual consta de normes i recomanacions específiques per Java per tal d'eliminar possibles vulnerabilitats en sistemes robustos; i CERT Perl, que està centrat en normes i recomanacions per evitar vulnerabilitats de llenguatge Perl.

CWE

CWE[13] (en anglès Common Weakness Enumeration) és una llista desenvolupada per la comunitat sobre tipus de debilitats en el programari que tenen relació amb problemes de seguretat. Aquestes debilitats són defectes o errors en la implementació, el codi, el disseny o l'arquitectura de programari que si no es resolen poden fer que els sistemes, les xarxes o el maquinari siguin vulnerables a atacs.

L'objectiu principal de CWE és aturar les vulnerabilitats a l'origen educant els i les professionals d'arquitectura de software, disseny, programació i adquirents sobre com eliminar els errors més comuns abans de lliurar els productes. L'ús de CWE ajuda a prevenir els tipus de vulnerabilitats de seguretat que poden afectar les indústries de programari i maquinari, avaluar la cobertura de les eines dirigides a aquestes debilitats, aprofitar un estàndard de referència comú per als esforços d'identificació, mitigació i prevenció de debilitats, i evitar les vulnerabilitats de programari i maquinari abans del desplegament.

5.4. Beneficis dels estàndards de codi en el desenvolupament de programari

En un equip de treball de persones programadores sense l'aplicació d'un codi establert, és molt comú que programin de manera individualitzada sense seguir un criteri únic, donant com a resultat un dipòsit difícil d'entendre i de mantenir. Un codi mal organitzat es pot transformar en un increment d'hores de treball per cercar l'origen d'un error, i crear divergències en l'equip de treball en el moment d'unificar la feina.

Implementar estàndards de codi en el desenvolupament de programari comporta beneficis com cultivar una cultura d'excel·lència, incrementar l'eficàcia i l'eficiència dels equips, minimitzar errors, reelaboracions i retards, i crear un bucle de retroalimentació positiva, tal com es detalla en els següents punts:

Cultiven una cultura d'excel·lència

Els estàndards de codi cultiven una cultura d'excel·lència en la qual els objectius i els resultats estan clars per a totes i tots els membres de l'equip. Són independents, documentats, àmpliament entesos i assolibles d'una manera clara i demostrable. Ja

[12] https://wiki.sei.cmu.edu/confluence/dashboard.action#all-updates
[13] https://cwe.mitre.org/index.html

no depèn de l'estil d'una persona o de la comprensió del que és ideal o de quina és la cultura d'equip actual.

Amb el temps, les pràctiques de desenvolupament de programari es tornen més eficients a mesura que l'equip aprèn a codificar l'estàndard d'una manera unificada i sistemàtica.

Incrementen l'eficàcia i eficiència

Les normes acceptades per la indústria no només identifiquen problemes a evitar, sinó que també proporcionen les millors pràctiques per tal d'ajudar a incrementar l'eficiència dels equips. A part d'identificar vulnerabilitats de seguretat i possibles defectes a evitar, aconsellen sobre estils i pràctiques recomanades. Totes les normes i recomanacions dels estàndards de codi continuen sent revisades, actualitzades i ampliades per cobrir nous problemes i escenaris (veure apartat 6.2).

Els equips de desenvolupament de programari poden beneficiar-se d'experiències prèvies en indústria i de la feina de milers d'experts per refinar les seves pràctiques i produir codi amb la més alta seguretat i fiabilitat possibles. Seguir aquestes millors pràctiques permet produir de manera més eficaç i ràpida, i evitar problemes.

Minimització d'errors, reelaboracions i retards

Quan s'està apunt de fer un llançament, sovint es descobreix un defecte o vulnerabilitat de seguretat que requereix inevitablement un retorn al desenvolupament i un reinici del procés per corregir el defecte abans de l'alliberament. Alternativament, també hi ha riscos que es lliuri a la clientela un programari amb un defecte, i un cop sigui identificat i analitzat, requereix que l'equip deixi les tasques d'aquell moment per arreglar i actualitzar el producte. Les conseqüències en una situació així poden ser encara més greus en el cas d'indústries crítiques, ja que poden ser molt visibles o poden causar problemes greus per a la seva seguretat.

En resum, el compliment total dels estàndards de codi evita costosos errors comuns durant tot el procés de desenvolupament, fet que al mateix temps evita reelaboracions i retards d'última hora.

Bucle de retroalimentació positiva

El compliment de les normes crea un bucle de retroalimentació positiva de defectes reduïts en el camp, un major avantatge competitiu amb una adherència als estàndards i una disminució del temps de comercialització de nous productes.

Els equips volen treballar en noves característiques innovadores per a productes dinàmics i en expansió. Comprometre's amb un estàndard de codi reconegut és un avantatge competitiu, ja que permet mantenir usuaris actuals i obrir portes als nous. A mesura que els problemes de programari i les vulnerabilitats de seguretat es fan més visibles, augmenta la inestabilitat i la inseguretat per part de les persones usuàries, i aquestes busquen reduir els riscos.

Quan es lliura un codi més fiable i compatible amb la indústria, amb menys defectes en el camp, l'equip pot oferir noves funcions i nous productes més ràpidament. Això, al mateix temps, millora la reputació de la marca i crea futures oportunitats. Així comença el bucle de retroalimentació.

Durant molts anys, la indústria dels dispositius mèdics va estar motivada per buscar estàndards de codi per aquesta mateixa raó, basant-se en gran mesura en els estàndards MISRA C (veure apartat 6.2.3). Amb el temps, s'ha vist l'aparició d'estàndards específics de la indústria com el reconeixement de la FDA d'UL2900 per a la seguretat en dispositius mèdics i xarxes IoT.

L'elecció de l'estàndard de codi adequat és important, ja que:

- Augmenta la fiabilitat del codi fent complir les regles.
- Educa les persones desenvolupadores sobre pràctiques segures de desenvolupament de codi, reduint el temps, els costos de la incorporació i la formació.
- Estableix un enfocament coherent per a l'anàlisi de codi, el qual es pot compartir entre equips.
- Proporciona flexibilitat per adaptar-se a les diferents necessitats, sense construir noves regles de desenvolupament de codi des de zero.

5.5. Principis SOLID

Avui dia, en la majoria de projectes de software d'alt nivell, degut a l'alt nivell d'abstracció, s'empra almenys en alguna part la programació orientada a objectes (POO). Per tal de crear un estàndard de bones pràctiques de codi que ajudessin a produir codi orientat a objectes de qualitat, als principis dels 2000 es van introduir els principis SOLID. Aquest acrònim mnemotècnic serveix per recordar els cinc principis que fan un programa orientat a objectes sòlids. Aquests principis són:

- **Responsabilitat única (single responsibility principle)**: un objecte només hauria de tenir un únic motiu per canviar.
- **Obert/Tancat (Open/Closed pr.)**: les entitats de software han d'estar obertes a la seva extensió i tancades a la seva modificació.
- **Principi de substitució de Liskov (Liskov sub. pr.)**: els objectes d'un programa haurien de ser substituïbles per instàncies dels seus subtipus sense alterar el correcte funcionament d'aquest programa.
- **Principi de segregació de la interfície (Interface seg. pr.)**: moltes interfícies de client específiques són millors que una de propòsit general.
- **Principi de inversió de la dependència (Dependency inv. pr.)**: la noció de que s'ha de dependre d'abstraccions, no d'implementacions. Per tal de seguir aquest principi s'empra la injecció de dependències.

L'assegurament de la qualitat és una manera sistemàtica de crear un entorn per garantir que el programari que s'està desenvolupant compleix els requisits de qualitat. Per tant, es tracta d'un procés preventiu que té com a objectiu establir la metodologia i l'estàndard correctes per oferir un entorn de qualitat al producte que s'està desenvolupant.

No hi ha una única manera de mesurar la qualitat del codi. El que es valora pot variar entre equips de desenvolupament, però alguns trets clau a mesurar per a una major qualitat del codi són [14]:

- **Funcionalitat**: eficàcia amb la qual el programari interactua amb altres components del sistema. El programari ha de proporcionar les funcions adequades segons els requisits, i aquestes funcions s'han d'implementar correctament.
- **Fiabilitat**: la fiabilitat és la capacitat del programari per funcionar en condicions específiques durant una durada especificada.
- **Usabilitat**: la usabilitat és la facilitat d'ús del programari, és a dir, facilitat amb què una persona usuària pot entendre les funcions del programari i esforços que es requereixen per seguir-ne les funcions.
- **Eficiència**: l'eficiència del programari depèn de l'arquitectura i la pràctica de desenvolupament de codi seguida durant el desenvolupament.
- **Manteniment**: el manteniment mesura la facilitat amb què es pot mantenir el programari. Es relaciona amb la mida, consistència, estructura i complexitat de la base de codi.
- **Portabilitat**: la portabilitat mesura com d'utilitzable és el mateix programari en diferents entorns. Es tracta de la independència de la plataforma, és a dir, la facilitat amb què un sistema s'adapta als canvis en les especificacions, com de fàcil és instal·lar el programari i com de fàcil és substituir un component en un entorn determinat.

6.1. Anàlisi estàtica de qualitat del codi

El desenvolupament de codi de qualitat comporta temps i esforç en els primers passos, però a llarg termini implica reduir el cost del manteniment i les correccions d'errors. Analitzar i mesurar la qualitat del codi pot ser complicat, ja que aquesta pot ser subjectiva, però es poden utilitzar algunes mètriques per avaluar-la objectivament, i hi ha diverses maneres de reduir-ne la complexitat i millorar-ne la qualitat.

L'anàlisi estàtica de codi identifica defectes, vulnerabilitats i problemes de compliment a mesura que es va codificant. Detecta problemes que sovint es perden quan s'utilitzen altres mètodes, com ara compiladors i revisions manuals de codi. Amb l'anàlisi estàtica de codi, els problemes de programari es poden solucionar abans, reduint els costos generals i permetent lliurar un producte de qualitat a temps. Consta d'una sèrie de comprovacions automatitzades que es realitzen en el codi font. Una eina d'anàlisi estàtica escaneja el codi a la recerca d'errors i vulnerabilitats comunes conegudes, com ara fugides de memòria o desbordaments de buffer[14]. L'anàlisi també pot fer complir les regles de desenvolupament de codi.

[14] Un desbordament de buffer o de la memòria intermèdia es produeix quan es col·loquen més dades en una memòria intermèdia de longitud fixa de les que poden gestionar. La informació addicional, que ha d'anar a algun lloc, pot desbordar-se a l'espai de memòria adjacent, corrompent o sobre-escrivint les dades contingudes en aquest espai.

Igual que totes les formes de proves automatitzades, l'anàlisi de codi estàtic garanteix que les comprovacions es realitzin de manera coherent i proporciona feedback ràpid sobre els últims canvis. No obstant això, només pot identificar els casos en els quals es trenquen les regles programades; no pot trobar totes les fallades únicament llegint el codi font. També existeix el risc de falsos positius, per la qual cosa és necessari interpretar els resultats.

En aquest sentit, l'anàlisi de codi estàtic és un complement valuós de les revisions de codi, ja que posa de manifest els problemes coneguts i allibera temps per a tasques com la revisió del disseny i l'enfocament general.

Els **beneficis** d'utilitzar l'anàlisi estàtica automatitzada inclouen:

- Processament de desenes de milers d'arxius en pocs minuts. Una persona revisora normalment només és capaç d'analitzar uns pocs centenars de línies de codi font per hora, la qual cosa redueix dràsticament el cicle de vida general del projecte i eleva la càrrega sobre els equips de treball.
- És menys costosa que les inspeccions manuals de codi. Identifica defectes coneguts sota demanda i mai es perd. Les inspeccions manuals de codi són menys efectives, consumeixen més temps i són més cares.
- Ofereixen resultats més precisos que una persona. Eliminen els errors i omissions que es produeixen durant les revisions manuals de codi, i per tant milloren la qualitat del codi.

Pel que fa als **inconvenients**, les organitzacions han d'estar al corrent dels següents:

- Es poden detectar falsos positius.
- És possible que una eina no indiqui quin és el defecte en el codi quan en troba un.
- No sempre es poden seguir totes les regles de desenvolupament de codi, com les regles que necessiten documentació externa.
- No pot detectar com s'executarà una funció.
- És possible que no es puguin analitzar les llibreries del sistema i de tercers.

6.2. Mètriques de qualitat de codi

La complexitat ciclomàtica i la complexitat de Halstead són dues mètriques quantitatives d'anàlisi estàtiques per a mesurar la qualitat de codi.

6.2.1. Complexitat ciclomàtica

La complexitat ciclomàtica (CC) és una mètrica que quantifica la qualitat de software, desenvolupada per Thomas J. McCade el 1976, que utilitza un càlcul senzill en base a un diagrama de flux de control representatiu del codi a analitzar [15]. Aquest diagrama de flux fa referència al nombre de camins linealment independents que el codi realitza al ser executat i es representa amb nodes i arestes. Cada node representa un bloc bàsic de codi i les arestes representen les connexions o camins entre blocs del codi del programa (Fig. 13).

A

```
int x,y,r;
if (x<0 || v<0) {
  system.out.println('X o Y son negatius');
} else {
  r=(x+y)/2;
  system.out.println('La mitjana de X i Y es:' + r ):
}
```

B

Figura 7. Exemple de càlcul de la Complexitat Ciclomàtica. A: fragment de codi a analitzar; B: esquema de nodes i arestes representatiu de l'execució del fragment de codi de la figura 7A. Els nodes blaus, representen l'entrada i la sortida de l'execució, els taronges les condicions, i les arestes les rutes d'execució.

El codi complex és poc fiable, ineficient i de baixa qualitat, per la qual cosa és important mesurar la complexitat ciclomàtica i així poder millorar la qualitat del codi si és necessari. Com més alt sigui el nombre de camins, més complex el codi, i més probable que contingui defectes i sigui difícil de testejar, de llegir i de mantenir.

Prenent d'exemple el fragment de codi de la figura 7A, el càlcul per mesurar la complexitat ciclomàtica (CC) és:

*CC = nº d'arestes – nº de nodes + 2*nº de nodes de sortida*

Com es pot observar en la figura 7B, els nodes blaus (2) representen l'entrada i la sortida de l'execució, els taronges (4) les condicions, i les arestes (7) les rutes d'execució. Per tant, la fórmula resultant seria:

$$CC = 7 - 6 + 2 = 3$$

Hi ha un acord en quant a la simplicitat d'un codi en funció del valor obtingut en el càlcul de la seva complexitat ciclomàtica. Segons el valor obtingut podem determinar si el codi és simple (valor CC entre 1-10), lleugerament complex (valor CC entre 11-20), complex (valor CC entre 21-50), o no testejable (valor CC de més de 50). Quan el codi té un valor entre 1 i 10 és considerat net, testejable, efectiu i gestionable. En canvi, per sobre de 10, el risc de defectes es va veient incrementat, fins arribar a 50, que es considera que no és testejable.

6.2.2. Complexitat de Halstead

La mesura de complexitat de Halstead es va desenvolupar per Maurice Halstead al 1977 com un mitjà per determinar una mesura quantitativa de complexitat a partir dels operadors i operands mitjançant uns indicadors [16]. Aquesta mètrica mesura la complexitat computacional del codi en termes d'error, dificultat i temps de prova.

Les mètriques de Halstead només tenen sentit al nivell de codi font, i varien amb els paràmetres següents:

PARÀMETRE	SIGNIFICAT
n_1	Nombre d'operadors diferents
n_2	Nombre d'operands diferents
N_1	Nombre d'instàncies de l'operador
N_2	Nombre d'instàncies d'operands

Taula 2. Paràmetres i significats de les mètriques Halstead.

I tenint en compte aquests paràmetres es poden mesurar diverses mètriques:

MÈTRICA	SIGNIFICAT	FÓRMULA
n	Vocabulari	n_1+n_2
N	Mida	N_1+N_2
V	Volum	$N*\log_2 n$
D	Dificultat	$(n_1/2)*(N_2/n_2)$
E	Esforç	V*D
B	Errors	V/3000
T	Temps de testatge	E/18

Taula 3. Mètriques, significats i fórmules de la complexitat de Halstead.

Prenent d'exemple el fragment de codi de la Figura 8, i entenent com a operador (n1): main, (), {}, int, scanf, &, =, ,, +, /, ;, printf; i com a operands (n2): x, y, z, avg, "%d %d %d", 3, "avg = %d"

```
main()
{
  int x, y, z, avg;
  scanf("%d %d %d", &x, &y, &z);
  avg = (x+y+z)/3;
  printf("avg = %d", avg);
}
```

Figura 8. Exemple de codi en C.

Les mètriques de Halstead es poden calcular així:

$$n_1 = 12,\ n_2 = 7;\ n = 19$$

$$N_1 = 27,\ N_2 = 15,\ N = 42$$

$$N = 12*\log_2 12+7*\log_2 7 = 62.7$$

$$V = 42*\log_2 19 = 178.4$$

$$D = (12/2)+(15/7) = 12.85$$

$$E = 12.85*178.4 = 2292.44$$

$$T = 2292.44/18 = 127.357 \text{ segons}$$

6.3. Proves

La fase de proves del cicle de vida del programari es fan per determinar si el disseny proposat compleix amb el conjunt inicial d'objectius, i busca detectar els errors comesos en les etapes anteriors per a corregir-los. Per suposat, l'ideal és fer-ho abans que la persona usuària final se'ls trobi. Existeixen diferents tipus de proves, cadascunes amb els seus objectius, focus i metodologies.

6.3.1. Proves unitàries

Tindran com a objectiu verificar la funcionalitat i estructura de cada component individual que compon el sistema. Aquest tipus de proves verificaran que:

- Els mòduls que componen el sistema estiguin lliures d'errors.
- La totalitat dels camins lògics principals s'executin correctament en cada mòdul de l'aplicació.
- La totalitat de les transaccions que realitza cada mòdul s'executen correctament.

6.3.2. Proves del sistema

Tindran com a objectiu la integració global, verificant el correcte funcionament de la totalitat de les interfícies entre els diferents mòduls que el componen i amb la resta de sistemes d'informació o eines amb les quals es comunicarà. Per això, s'executaran el següent llistat de proves:

- Proves funcionals, amb l'objectiu d'assegurar que el sistema realitza correctament totes les funcions que s'han detallat en les especificacions donades per la persona usuària del sistema.
- Proves d'integració, per validar que el sistema es comunica correctament amb aplicacions de tercers. Dins d'aquesta mena de proves, es realitzaran les proves de les interfícies home-màquina.
- Proves de rendiment, amb l'objectiu de verificar que els temps de resposta es troben dins dels intervals establerts en les especificacions del sistema.
- Proves de volum, en les quals es monitorarà el funcionament del sistema quan es trobi treballant amb grans volums de dades. Principalment es faran proves en els mòduls.
- Proves de sobrecàrrega, per comprovar el funcionament del sistema quan està sotmès a carregues massives, amb l'objectiu de definir els llindars màxims d'operació en els quals el sistema opera per sota dels requisits establerts.
- Proves de còpia de seguretat, amb l'objectiu de verificar que el sistema pot recuperar-se davant fallades, tant d'equip físic com lògic, sense comprometre la integritat de les dades.
- Proves de facilitat d'ús, per comprovar l'adaptabilitat del sistema a les necessitats de la persona usuària en matèria d'usabilitat del sistema.
- Proves d'operació, amb l'objectiu de verificar la correcta implementació dels procediments d'operació, incloent-hi la planificació i control de treballs, arrencada i reinici del sistema.
- Proves de seguretat del sistema, per verificar els mecanismes de control d'accés al sistema.

6.3.3.

Proves d’implantació

Es realitzaran proves d’implantació amb l’objectiu de verificar el correcte funcionament de la totalitat dels mòduls i funcionalitats del sistema, estant aquest totalment integrat tant en maquinari com en programari, a l’entorn d’operació final. Per això, seran les persones usuàries qui, des d’un punt de vista d’operació, realitzin l’acceptació del sistema una vegada implementat sobre l’entorn real d’operació. Destacar que, en aquest punt, també es realitzaran proves de copia de seguretat del sistema amb l’objectiu de verificar que no es veu compromès el seu funcionament en existir un control i seguiment dels procediments de salvaguarda i de recuperació de la informació.

6.3.4.

Proves d’acceptació

Conjunt de proves que es realitzaran al sistema amb l’objectiu de validar que compleix amb el funcionament esperat, en matèria de funcionalitat i rendiment. Aquestes proves seran definides i executades pels usuaris i usuàries del sistema, que seran les persones responsables de la seva acceptació.

6.3.5.

Proves de regressió

Les proves de regressió es realitzaran sempre que s’incorpori un nou mòdul o funcionalitat al projecte, amb l’objectiu de verificar que els canvis implementats no introdueixen comportaments no desitjats en el sistema. Per a l’execució d’aquestes, serà necessari repetir la bateria de proves definides anteriorment.

7. Certificacions ISO

L'ISO, també coneguda com a Organització Internacional per a la Normalització[15], és una organització internacional independent i no governamental que, a través dels seus membres, reuneix persones expertes per compartir coneixements i desenvolupar estàndards internacionals voluntaris que cobreixen aspectes de tecnologia i fabricació per garantir la qualitat, la salut, el servei al client, la seguretat i la protecció de dades.

L'objectiu principal de la ISO és ajudar les empreses a optimitzar els seus processos per tal d'augmentar la qualitat dels seus serveis, proporcionant un conjunt de requisits i normes a seguir. Tal com s'ha vist al llarg del document, un estàndard es descriu com un mètode de millor pràctica avalat per experts, ja sigui per al desenvolupament d'un programari, per a la realització d'una avaluació de riscos o per una altra activitat.

Les normes internacionals permeten als consumidors confiar que els productes són fiables i de bona qualitat. El certificat ISO és un reconeixement internacional que ofereix a les empreses la possibilitat d'operar més enllà de les fronteres nacionals, permetent augmentar vendes i ingressos. Les etiquetes ISO ajuden a millorar la imatge de les empreses, ja que aquestes poden demostrar que treballen conforme les normes internacionals. Per a molts compradors i clients, és un senyal que les empreses ofereixen productes i serveis excellents.

Les normes ISO intenten garantir la qualitat, la coherència i la seguretat. Hi ha múltiples beneficis per a les entitats que compleixen aquestes normes, com per exemple la fiabilitat, la millora del rendiment i de qualitat, la reducció de risc, la sostenibilitat i la innovació. Desenvolupar un programari sota les normes ISO garanteix una millor qualitat i robustesa del codi disminuint la probabilitat d'errors en producció.

ISO aplicables al desenvolupament de codi

A continuació s'expliquen algunes de les normes ISO de desenvolupament de programari:

- **ISO 27000**: Seguretat de la informació
- **ISO 29119**: Proves de programari
- **ISO 25001**: Sistemes i requisits de qualitat i avaluació de programari (SQuaRE)

[15] https://www.iso.org/

Un altre estàndard és la família ISO 27000, que inclou els estàndards de seguretat de la informació dins d'una organització. L'objectiu principal d'aquesta ISO és protegir els actius de l'empresa i millorar les seves pràctiques de seguretat.

L'ISO 27001 és l'estàndard reconegut internacionalment per als sistemes de gestió de la seguretat de la informació (ISMS), i consta de polítiques, processos, controls i altres components que impliquen persones de l'empresa, així com processos, propietats i infraestructura tecnològica utilitzada per l'organització.

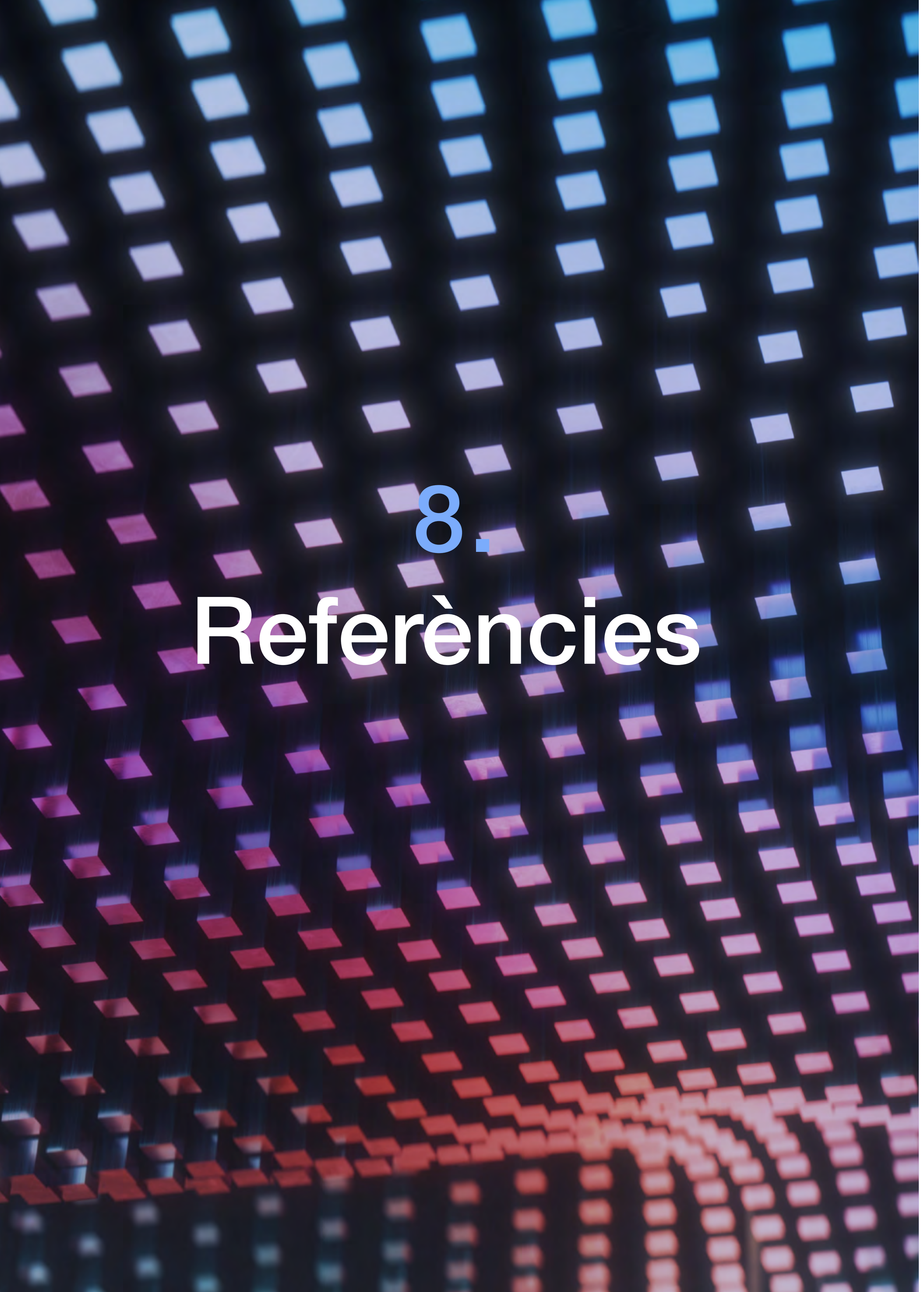
L'objectiu principal d'aquesta norma és garantir que la informació i les instal·lacions de processament d'informació siguin segures i que el compliment legal s'implementi i es mantingui correctament a totes les organitzacions. També utilitza processos de gestió de riscos i oportunitats, avaluacions de riscos i tractaments de risc per mitigar i respondre adequadament a les amenaces i incidents de seguretat, com ara delictes cibernètics, incompliment de dades personals, danys, mal ús o atacs virals. ISO 27001 inclou controls per protegir la informació d'identificació personal que ajuden al compliment del RGPD i com a mètode per aconseguir una bona seguretat de les dades.

La ISO 29119 se centra en les proves de programari. La idea principal d'aquesta ISO és que les proves són l'enfocament principal per a la mitigació i la prevenció de riscos. Per tant, tots els estàndards segueixen l'enfocament basat en el risc i animen a les empreses a centrar-se en les funcions més importants.

La ISO/IEC 25001:2014 proporciona requisits i recomanacions per a una entitat responsable d'implementar i gestionar els sistemes i les activitats d'especificació i avaluació de requisits de qualitat dels productes de programari a través de la provisió de tecnologia, eines, experiències i habilitats de gestió.

Beneficis:

- Assegurar la reducció de fallades en el programari després de la seva implantació en producció.
- Avaluar i controlar el rendiment del producte de programari desenvolupat, assegurant que podrà generar els resultats tenint en compte les restriccions de temps i recursos establertes.
- Assegurar que el producte de programari desenvolupat respecta els nivells necessaris per a les característiques de seguretat (confidencialitat, integritat, autenticitat, etc.).
- Comprovar que el producte desenvolupat podrà ser posat en producció sense posar en compromís la resta de sistemes i mantenint la compatibilitat amb les interfícies necessàries.



8. Referències

[1] Index, T. I. O. B. E. (2018). Tiobe-the software quality company. TIOBE Index| TIOBE–The Software Quality Company [Electronic resource]. https://www. tiobe. com/tiobe-index/

[2] Fangohr, H. (2004). A Comparison of C, MATLAB, and Python as Teaching Languages in Engineering. In: Bubak, M., van Albada, G.D., Sloot, P.M.A., Dongarra, J. (eds) Computational Science - ICCS 2004. ICCS 2004. Lecture Notes in Computer Science, vol 3039. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-25944-2_157

[3] Baskova, O., Gatsenko, O., & Gordienko, Y. (2010). Enabling high-performance distributed computing to e-science by integration of 4th generation language environments with desktop grid architecture and convergence with global computing grid. In Proc. of Cracow Grid Workshop (CGW’10) (pp. 234-243).

[4] Ogala, J. O., & Ojie, D. V. (2020). Comparative analysis of c, c++, c# and java programming languages. GSJ, 8(5).

[5] A. Dinh, S. Miertschin, A. Young, and S. D. Mohanty, “A data-driven approach to predicting diabetes and cardiovascular disease with machine learning,” BMC Medical Informatics and Decision Making, vol. 19, no. 1, Nov. 2019, doi: 10.1186/s12911-019-0918-5.

[6] Takeuchi, H., & Nonaka, I. (1986). The new new product development game. Harvard business review, 64(1), 137-146.

[7] Sutherland, J. (1993). The Plan is the Problem!.

[8] Sutherland, J. V., & Schwaber, K. (1995). The SCRUM methodology. In Business object design and implementation: OOPSLA workshop.

[9] S. Al-Saqqa, S. Sawalha i H. AbdelNabi, «Agile Software Development: Methodologies and Trends,» International Journal of Interactive Mobile Technologies, pp. vol. 14, nº 11, 2020.

[10] H. Wickham, Advanced R, 2nd ed. Chapman and Hall/CRC, 2019.

[11] K. Reitz, The Hitchhiker’s Guide to Python. O’Reilly Media, 2016. [Online]. Available: https://docs.python-guide.org/

[12] MISRA C, Guidelines for the use of the C language in critical systems. HORIBA MIRA Limited, 2020.

[13] MISRA C++, Guidelines for the use of the C++ language in critical systems. HORIBA MIRA Limited., 2020.

[14] “What is Quality Assurance (QA): Tutorial, Attributes, Components, Types.” https://www.javatpoint.com/quality-assurance

[15] T. J. McCabe, “A Complexity Measure,” IEEE Transactions on Software Engineering, vol. SE-2, no. 4, pp. 308–320, Dec. 1976, doi: 10.1109/tse.1976.233837.

[16] T. Hariprasad, G. Vidhyagaran, K. Seenu, and C. Thirumalai, “Software complexity analysis using halstead metrics,” May 2017, doi: 10.1109/icoei.2017.8300883.

ANNEX 1: Tidyverse Style Guide

1. Fitxers

1.1 Noms

Els noms dels fitxers han de tenir un significat i acabar en .R. Evitar utilitzar caràcters especials als noms dels fitxers: utilitzar números, lletres i "_".

1.2 Organització

Donar un nom concís a un fitxer ajuda a una bona organització.

1.3 Estructura interna

Utilitzar línies comentades de "-" i "=" per dividir el fitxer en fragments fàcilment llegibles.

2. Sintaxi

2.1 Noms d'objectes

Els noms de variables i funcions només han d'utilitzar lletres minúscules, números i "_". Utilitzar barra baixa (_) per separar paraules dins d'un nom (exemple "dia_1" en lloc de "dia1").

2.2. Espaiat

2.2.1 Comes

Posar sempre un espai després d'una coma, mai abans.

2.2.2 Parèntesis

No col·locar espais entre parèntesis dins o fora per a execucions de funcions regulars.

Col·locar un espai abans i després () quan s'utilitzi if, for o while.

Col·locar un espai després de () quan es vagi a escriure els arguments de la funció.

2.2.3 Símbols clau

Els parèntesi clau {{ }} sempre han de tenir espais interiors que ajudin a emfatitzar el seu comportament específic.

2.2.4 Insertar informació

Els símbols d'insertar informació (==, +, -, <-, etc.) sempre han d'estar envoltats d'espais.

2.2.5 Espais addicionals

Afegir espais addicionals està bé si millora l'alineació de = o <-.

2.3 Funcions

2.3.1 Arguments

Els arguments d'una funció solen dividir-se en dues grans categories: una proporciona les dades per calcular i l'altra controla els detalls del càlcul. Quan s'anomena una funció, normalment s'ometen els noms dels arguments de dades ja que s'utilitzen comunament. Si se sobreescriu el valor per defecte d'un argument, s'ha d'utilitzar el nom complet.

2.3.2 Assignació

Evitar l'assignació de variables dins de funcions.

2.4 Control del flux

2.4.1 Blocs de codi

Els símbols {} defineixen la jerarquia més important del codi R. Per facilitar aquesta jerarquia:

- { ha de ser l'últim caràcter de la línia. El codi ha d'estar en la mateixa línia que el símbol { d'obertura.
- El contingut ha d'estar espaiat per dos espais.
- } ha de ser el primer símbol de la línia.

2.4.2 Funcions if

- Si s'utilitza, else ha d'estar en la mateixa línia que el símbol }.
- & i | mai s'ha d'utilitzar dins d'una clàusula if perquè poden retornar vectors. S'ha d'utilitzar && i || .
- ifelse(x, a, b) no es substitut directe de if (x) a else b.

2.4.3 Sentència en línia

S'accepta no utilitzar els caràcters {} per execucions senzilles que es poden escriure en una línia, sempre i quan aquesta no tingui efectes secundaris en la resta de codi. Les funcions return(), stop() or continue s'han d'executar en el seu bloc {} específic.

2.4.4 Coacció implícita

Evitar la coacció implícita en les funcions if (per exemple de valor numèric a lògic).

2.4.5 Funció switch()

- Evitar utilitzar la funció switch() en cada posició.
- Cada element ha d'estar a la seva línia.
- Els elements que van al següent element han de tenir un espai després de =.
- Proporcionar una opció d'error de sortida, tret que s'hagi validat prèviament l'entrada.

2.5 Línies llargues

Limitar el codi a 80 caràcters per línia.

2.6 Punt i coma

No posar ; al final d'una línia, i no l'utilitzar ; per posar diverses ordres en una línia.

2.7 Assignació de variables

Utilitzar <-, i no =, per a l'assignació.

2.8 Dades

2.8.1 Caràcters

Utilitzar " " " i no " ' " per comentar text. L'única excepció és quan en el text doble comes.

2.8.2 Definicions lògiques

Es prefereix TRUE I FALSE enlloc de T i F.

2.9 Comentaris

Cada línia de comentari ha de començar amb el símbol del comentari i un únic espai: #

3. Funcions

3.1 Nomenar

Utilitzar verbs per als noms de funcions seguint les indicacions explicades al punt 2.1.

3.2 Línies llargues

Hi ha dues opcions si el nom de la funció i la definició no poden cabre en una sola línia

- Col·locar cada argument en la seva pròpia línia de codi, posant el primer seguidament de l'obertura de la funció.
- Col·locar cada argument en la seva pròpia línia de codi, posant el primer en la línia següent de l'obertura de la funció.

3.3 return()

Utilitzar return() només per a retorns ràpids. En cas contrari, s'ha de dependre de R per retornar el resultat de l'última expressió avaluada.

3.4 Comentaris

En codi, utilitzar comentaris per explicar el "per què" i no el "què" o el "com". Cada línia d'un comentari ha de començar amb el símbol del comentari i un únic espai: #.

4. Pipes

4.1 Introducció

Utilitzar %>% per emfatitzar una seqüència d'accions, en lloc de l'objecte en què s'estan duent a terme les accions.

Eviteu utilitzar l'opció pipe quan:

- S'ha de manipular més d'un objecte a la vegada. Pipe es reserva per a una seqüència de passos aplicats a un objecte primari.
- Hi ha objectes intermedis significatius que podrien rebre noms informatius.

4.2 Espais en blanc

%>% sempre ha de tenir un espai abans, i normalment ha de ser seguit per una nova línia. Després del primer pas, cada línia ha d'anar seguit de dos espais. Aquesta estructura fa que sigui més fàcil afegir nous passos (o reorganitzar els passos existents) i és més difícil passar per alt un pas.

4.3 Línies llargues

Si els arguments d'una funció no encaixen tots en una línia, cada argument ha d'anar a la seva pròpia línia i posició.

4.4 Pipes curts

Quan és un sol pas es pot escriure en una sola línia de codi. Tot i així, si s'ha d'ampliar en un futur és millor escriure-ho en una funció regular.

4.5 Sense arguments

magrittr permet ometre () en funcions que no tenen arguments. S'ha d'evitar.

4.6 Assignació

Hi ha tres formes acceptables d'assignació de variables:

- Nom de la variable i assignació en línies separades
- Nom i assignació de variables en la mateixa línia
- Assignació al final d'una pipe amb ->

ANNEX 2: Guia de bones pràctiques de programació en R de Google

1. Fitxers

1.1 Noms

Els noms dels fitxers han de tenir un significat i acabar en .R. Evitar utilitzar caràcters especials als noms dels fitxers: utilitzar números, lletres i "_".

S'ha de documentar tot.

1.2 Organització

Donar un nom concís a un fitxer ajuda a una bona organització.

1.3 Estructura interna

Utilitzar línies comentades de "-" i "=" per dividir el fitxer en fragments fàcilment llegibles.

2. Sintaxi

2.1 Noms d'objectes

2.1.1 CamelCase

Els noms de les funcions tenen majúscules inicials (CamelCase)

2.1.2 Punt a inici de funció privada

El noms de les funcions privades tenen un punt a l'inici.

2.2. Espaiat

2.2.1 Comes

Posar sempre un espai després d'una coma, mai abans.

2.2.2 Parèntesis

No col·locar espais entre parèntesis dins o fora per a execucions de funcions regulars.

Col·locar un espai abans i després () quan s'utilitzi if, for o while.

Col·locar un espai després de () quan es vagi a escriure els arguments de la funció.

2.2.3 Símbol clau

Els parèntesis clau {{ }} sempre han de tenir espais interiors que ajudin a emfatitzar el seu comportament específic.

2.2.4 Insertar informació

Els símbols d'insertar informació (==, +, -, <-, etc.) sempre han d'estar envoltats d'espais.

2.2.5 Espais addicionals

Afegir espais addicionals està bé si millora l'alineació de = o <-.

2.3 Funcions

2.3.1 Arguments

Els arguments d'una funció solen dividir-se en dues grans categories: una proporciona les dades per calcular i l'altra controla els detalls del càlcul. Quan s'anomena una funció, normalment s'ometen els noms dels arguments de dades ja que s'utilitzen comunament. Si se sobreescriu el valor per defecte d'un argument, s'ha d'utilitzar el nom complet.

2.3.2 Assignació

Evitar l'assignació de variables dins de funcions.

2.3.3 No utilitzar attach()

Es recomana no utilitzar attach() ja que la possibilitat de crear errors és molt elevada.

2.4 Control del flux

2.4.1 Blocs de codi

Els símbols {} defineixen la jerarquia més important del codi R. Per facilitar aquesta jerarquia:

- { ha de ser l'últim caràcter de la línia. El codi ha d'estar en la mateixa línia que el símbol { d'obertura.
- El contingut ha d'estar espaiat per dos espais.
- } ha de ser el primer símbol de la línia.

2.4.2 Funcions if

- Si s'utilitza, else ha d'estar en la mateixa línia que el símbol }.
- & i | mai s'ha d'utilitzar dins d'una clàusula if perquè poden retornar vectors. S'ha d'utilitzar && i || .
- ifelse(x, a, b) no és substitut directe de if (x) a else b.

2.4.3 Sentència en línia

S'accepta no utilitzar els caràcters {} per execucions senzilles que es poden escriure en una línia, sempre i quan aquesta no tingui efectes secundaris en la resta de codi. Les funcions return(), stop() or continue s'han d'executar en el seu bloc {} específic.

2.4.4 Coacció implícita

Evitar la coacció implícita en les funcions if (per exemple de valor numèric a lògic).

2.4.5 Funció switch()

- Evitar utilitzar la funció switch() en cada posició.
- Cada element ha d'estar a la seva línia.
- Els elements que van al següent element han de tenir un espai després de =.
- Proporcionar una opció d'error de sortida, tret que s'hagi validat prèviament l'entrada.

2.5 Línies llargues

Limitar el codi a 80 caràcters per línia.

2.6 Punt i coma

No posar ; al final d'una línia, i no l'utilitzar ; per posar diverses ordres en una línia.

2.7 Assignació de variables

Utilitzar <-, i no =, per a l'assignació.

2.8 Dades

2.8.1 Caràcters

Utilitzar “ “ ” i no “ ‘ ” per comentar text. L'única excepció és quan en el text hi ha doble comes.

2.8.2 Definicions lògiques

Es prefereix TRUE I FALSE enlloc de T i F.

2.9 Comentaris

Cada línia de comentari ha de començar amb el símbol del comentari i un únic espai: #

3. Funcions

3.1 Nomenar

Utilitzar verbs per als noms de funcions seguint les indicacions explicades al punt 2.

3.2 Línies llargues

Hi ha dues opcions si el nom de la funció i la definició no poden cabre en una sola línia

- Col·locar cada argument en la seva pròpia línia de codi, posant el primer seguidament de l'obertura de la funció.
- Col·locar cada argument en la seva pròpia línia de codi, posant el primer en la línia següent de l'obertura de la funció.

3.3 return()

Utilitzar sempre return().

3.4 Comentaris

En codi, utilitzar comentaris per explicar el “per què” i no el “què” o el “com”. Cada línia d'un comentari ha de començar amb el símbol del comentari i un únic espai: #.

4. Pipes

4.1 Introducció

Utilitzar %>% per emfatitzar una seqüència d'accions, en lloc de l'objecte en què s'estan duent a terme les accions.

Eviteu utilitzar l'opció pipe quan:

- S'ha de manipular més d'un objecte a la vegada. Pipe es reserva per a una seqüència de passos aplicats a un objecte primari.
- Hi ha objectes intermedis significatius que podrien rebre noms informatius.

4.2 Espais en blanc

%>% sempre ha de tenir un espai abans, i normalment ha de ser seguit per una nova línia. Després del primer pas, cada línia ha d'anar seguit de dos espais. Aquesta estructura fa que sigui més fàcil afegir nous passos (o reorganitzar els passos existents) i és més difícil passar per alt un pas.

4.3 Línies llargues

Si els arguments d'una funció no encaixen tots en una línia, cada argument ha d'anar a la seva pròpia línia i posició.

4.4 Pipes curts

Quan és un sol pas es pot escriure en una sola línia de codi. Tot i així, si s'ha d'ampliar en un futur és millor escriure-ho en una funció regular.

4.5 Sense arguments

magrittr permet ometre () en funcions que no tenen arguments. S'ha d'evitar.

4.6 Assignació

Hi ha dues formes acceptables d'assignació de variables:

- Nom de la variable i assignació en línies separades
- Nom i assignació de variables en la mateixa línia
- Assignació al final d'una pipe amb -> no és acceptable

ANNEX 3: Guia de bones pràctiques de programació en R d'Amazon AWS

La Guia de bones pràctiques d'Amazon per programar en R es basa en la guia d'estil R de Google (explicada en l'Annex 2) i “Advanced R, de Hadley Wickham”, amb alguns ajustaments explicats a continuació.

1. Noms i anotació

1.1 Noms de fitxer

Els noms dels fitxers han de tenir un significat i acabar en .R.

Si els fitxers s'han d'executar en seqüència, s'han d'afegir un prefix amb números en cada un d'ells.

1.2 Noms d'objecte

Els noms de variables i funcions han d'estar en minúscula.

Utilitzar un subratllat (_) per separar paraules dins d'un nom.

Els noms de variables han de ser substantius i els noms de funcions han de ser verbs.

Buscar noms concisos i amb cert significat.

Sempre que sigui possible, evitar utilitzar noms de funcions i variables ja existents. Fer-ho causarà confusió als lectors del codi.

2. Sintaxi

2.1 Espaiat

Col·locar espais al voltant de tots els operadors infix (=, +, -, <-, etc.). La mateixa regla s'aplica quan s'utilitza = en funcions. Posar sempre un espai després d'una coma, i mai abans.

Col·locar un espai a l'esquerra abans del parèntesi, excepte en funcions.

L'espaiat addicional (és a dir, més d'un espai seguit) és acceptable si millora l'alineació de signes o assignacions iguals (<-).

No col·locar espais al voltant del codi entre parèntesis o claudàtors.

2.2 Símbol clau {}

L'inici d'un parèntesi clau mai ha d'anar en la seva pròpia línia i sempre ha d'anar seguit d'una nova línia. El tancament sempre ha d'anar en la seva pròpia línia, llevat que sigui seguit per una altra.

2.3 Longitud de la línia

Limitar el codi a 80 caràcters per línia. Si no hi ha espai és indicació que s'ha de comprimir part del codi o separar en diferents funcions.

2.4 Sagnia

Quan se sagni el codi, utilitzar dos espais. No utilitzeu mai tabuladors ni barreja d'espais i tabuladors.

L'única excepció és si una definició de funció passa per sobre de diverses línies. En aquest cas, sagna la segona línia cap a on comença la definició.

2.5 Assignació

Utilitzeu <-, mai =, per a l'assignació.

3. Organització

3.1 Directrius de comentaris

Comentar el codi. Cada línia d'un comentari ha de començar amb el símbol del comentari i un únic espai: #.

Utilitzar comentaris per explicar el "per què" i no el "què" o el "com".

Els comentaris curts es poden col·locar després del codi precedit per dos espais, #, i després un espai.

Utilitzar línies comentades de - i = per dividir el fitxer en trossos fàcilment llegibles.

3.2 Definicions de funcions i execució de funcions

Les funcions primer han de llistar arguments sense valors per defecte, seguidament d'aquells arguments amb valors per defecte.

Tant en definicions de funcions com en execució, es permeten múltiples arguments per línia. Només es permeten salts de línia entre tasques.

3.3 Documentació de la funció

Les funcions han de tenir una secció de comentaris immediatament per sota de la línia de la funció. Aquests comentaris han de consistir en:

- Una descripció d'una frase de la funció.
- Una llista dels arguments de la funció, indicat amb "Args:" seguit d'una descripció de cadascun (inclòs el tipus de dades).
- I una descripció del valor de sortida, indicat amb "Returns:".

Els comentaris han de ser prou descriptius perquè una persona pugui utilitzar la funció sense llegir cap codi per entendre-la.

ANNEX 4: Guia de bones pràctiques per Python – PEP 8

1. Establiment del codi

1.1 Sagnat

Utilitzar 4 espais per nivell de sagnat.

1.2 Tabulació i espais

Els espais són el mètode preferit de sagnat. Les tabulacions s'han d'utilitzar únicament per mantenir-se coherents amb el codi que ja està sagnat amb tabulacions. Python no permet barrejar espais ni tabulacions.

1.3 Longitud màxima de la línia

Limitar totes les línies a un màxim de 79 caràcters.

1.4 Salt de línia

Es recomana fer un salt de línia abans de l'operador quan s'escriuen fórmules enlloc de fer el salt de línia després de l'operador, tot i que encara s'accepten les dues opcions.

1.5 Línies en blanc

- Envoltar les definicions de funció i classe de nivell superior amb dues línies en blanc.
- Les definicions de mètode dins d'una classe estan envoltades per una sola línia en blanc.

1.6 Codificació de fitxers d'origen

El codi sempre ha d'utilitzar UTF-8, i no ha de tenir una declaració de codificació. A la llibreria estàndard, les codificacions no UTF-8 s'han d'utilitzar només per a test.

1.7 Importacions

- Les importacions normalment han d'estar en línies separades.
- Les importacions sempre es col·loquen a la part superior del fitxer, just després de qualsevol comentari de mòdul i docstrings, i abans de qualsevol mòdul i constant.
- Es recomanen importacions totals, ja que solen ser més llegibles i tendeixen a comportar-se millor (o almenys donar millors missatges d'error) si el sistema d'importació està configurat incorrectament.

1.8 Dunder mòduls

Els mòduls dunders (noms amb dos barres baixes a banda i banda, __nom__) s'han de col·locar després del mòdul docstring, però abans de qualsevol declaració d'importació.

2. String quotes

És indiferent utilitzar cometes simples o dobles en strings. Però si es comença amb un tipus s'ha de seguir amb el tipus escollit.

3. Espais en blanc en expressions i declaracions

3.1 Pet Peeves

Evitar espais en blanc addicionals en les situacions següents:

- Immediatament dins de parèntesis o claudàtors.
- Entre una coma final i el següent parèntesi.
- Immediatament abans d'una coma, dos punts o punt i coma.
- No obstant això, els dos punts actuen com un operador binari, i han de tenir les mateixes quantitats a banda i banda (tractant-lo com l'operador amb la prioritat més baixa). En un codi més extens, els dos punts han de tenir el mateix espaiat aplicat.
- Immediatament abans del parèntesi d'inici d'una llista d'arguments d'execució d'una funció.
- Immediatament abans del parèntesi d'inici d'una indexació.
- Més d'un espai en l'assignació d'un operador per alinear-lo amb un altre.

3.2 Altres recomanacions

- Evitar els espais en blanc finals en qualsevol lloc. Com que normalment són invisibles, poden ser confusos.
- Envoltar sempre aquests operadors binaris amb un sol espai a banda i banda: assignació (=), assignació augmentada (+=, -= etc.), comparacions (==, <, >, !=, <>, <=, >=, in, not in, is, is not), booleans (and, or, not).
- Si s'utilitzen operadors amb prioritats diferents, considerar afegir espais en blanc al voltant dels operadors amb la prioritat més baixa. Mai utilitzar més d'un espai i sempre tenir la mateixa quantitat d'espais en blanc a banda i banda d'un operador binari.
- Les anotacions de funcions han d'utilitzar les regles normals per als dos punts i sempre tenir espais al voltant de la fletxa "->" si estan presents.
- No utilitzar espais al voltant del símbol "=" quan s'utilitza per indicar un argument de paraula clau, o quan s'utilitza per indicar un valor per defecte per a un paràmetre de funció no anotat.
- Tot i que de vegades està bé posar en la mateixa línia "if/for/while" quan és un codi curt, no s'ha de fer mai quan hi ha diverses clàusules en el codi.

4. Quan utilitzar comes finals

Les comes posteriors solen ser opcionals, però són obligatòries a l'hora de fer una tupla d'un element. Per a més claredat, es recomana posar aquest últim entre parèntesis (tècnicament redundant).

5. Comentaris

Posar explicacions de codi comprensibles. Els comentaris que contradiuen el codi són pitjors que cap comentari. Cal prioritzar el fet de mantenir els comentaris actualitzats a mesura que el codi canvia.

5.1 Comentaris de bloc

- Els comentaris de bloc generalment s'apliquen a algun codi que els segueix, i estan escrits al mateix nivell que aquest codi.
- Cada línia d'un comentari de bloc ha de començar amb un # i un sol espai (llevat que hi hagi text sagnat dins del comentari).
- Els paràgrafs dins d'un comentari de bloc han d'estar separats per una línia que conté un sol #.

5.2 Comentaris en línia

Utilitzar els comentaris en línia amb moderació.

5.3. Cadenes de documentació

Les convencions per escriure bones documentacions de strings (docstrings) es poden consultar a PEP 257[16].

6. Convencions de nomenclatura

6.1 Principi de substitució

Els noms que són visibles per a la persona usuària com a parts públiques de l'API han de seguir convencions que reflecteixin l'ús en lloc de la implementació.

6.2 Noms a evitar

No utilitzar mai els caràcters "l" (la lletra ela), "O" o "I" (i majúscula) com a noms de variables senzilles.

6.3 Compatibilitat ASCII

Els identificadors utilitzats a la llibreria estàndard han de ser compatibles amb ASCII tal com es descriu a la secció de polítiques de PEP 3131[17].

6.4 Noms de paquets i mòduls

Els mòduls han de tenir noms curts i en minúscules. Les barres baixes es poden utilitzar al nom del mòdul si això millora la llegibilitat. Els paquets de Python també han de tenir noms curts i en minúscules, tot i que es desaconsella l'ús de barres baixes.

6.5 Noms de classe

Els noms de classe normalment han d'utilitzar la convenció de CamelCase.

6.6 Noms d'excepció

S'ha d'utilitzar el sufix "Error" als noms d'excepció (si l'excepció és realment un error).

6.7 Noms de variables globals

Les normes són aproximadament les mateixes que les de les funcions.

6.8 Noms de funcions i variables

[16] https://peps.python.org/pep-0257/
[17] https://peps.python.org/pep-3131/

- Els noms de les funcions han de ser en minúscula, amb paraules separades per barres baixes segons sigui necessari per millorar la llegibilitat.
- Els noms de variables segueixen la mateixa norma que els noms de les funcions.

6.9 Arguments de funció i mètode

Utilitzar sempre “self” per al primer argument en mètodes d’instància.

Utilitzar sempre “cls” per al primer argument en mètodes de classe.

6.10 Noms de mètodes i variables d’instància

- Utilitzar les regles de nomenclatura de funcions: minúscules amb paraules separades per subratllar segons sigui necessari, per tal de millorar la llegibilitat.
- Utilitzar un subratllat inicial només per a mètodes no públics i variables d’instància.

6.11 Constants

Les constants es defineixen generalment a nivell de mòdul i s’escriuen amb totes les lletres majúscules i amb barres baixes separant paraules.

6.12 Disseny per a l’herència

Decidir si els mètodes i les variables d’una classe (col·lectivament: atributs) han de ser públiques o no. En cas de dubte, triar no públic; és més fàcil fer-ho públic més tard que fer que un atribut públic deixi de ser-ho.

7. Recomanacions de programació

- El codi s’ha d’escriure d’una manera que no desfavoreixi altres implementacions de Python (PyPy, Jython, IronPython, Cython, Psyco, etc).
- Les comparacions amb singletons com “None” sempre s’han de fer amb “is” o “is not”, mai amb els operadors d’igualtat.
- Utilitzar l’operador “is not” en lloc de “not … is”.
- Utilitzar sempre una sentència “def” en lloc d’una sentència d’assignació que vinculi una expressió lambda directament a un identificador.
- Utilitzar l’encadenament d’excepcions adequadament.
- En detectar errors del sistema operatiu, utilitzar la jerarquia d’excepcions explícita introduïda a Python 3.3 abans que la introspecció dels valors “errno”.
- Per a totes les clàusules de “try”/”except”, limitar la clàusula de “try” a la quantitat mínima de codi necessària. Això evita emmascarar errors.
- Quan un recurs és específic a una secció de codi en particular, utilitzar “with” per assegurar-se que es neteja de manera ràpida i fiable després del seu ús.
- Els gestors de context s’han d’utilitzar a través de funcions o mètodes separats sempre que facin alguna cosa que no sigui adquirir i alliberar recursos.
- Ser coherents en les declaracions de “return”. Totes les declaracions de retorn d’una funció han de retornar una expressió.
- Utilitzar ‘’.startswith() i ‘’.endswith() en lloc de tallar cadenes per comprovar si hi ha prefixos o sufixos. startswith() i endswith() són més nets i menys propensos a errors.
- Les comparacions entre objectes sempre han d’utilitzar isinstance() en lloc de comparar tipus directament.
- Per a seqüències (strings, llistes, tuples), tenir en compte que les seqüències buides són falses.
- No escriure strings que depenguin d’espais en blanc al final. Aquests espais en blanc finals són visualment indistingibles i alguns programadors els retallaran.
- No comparar els valors booleans amb True o False utilitzant “==”.

ANNEX 5: Guia de bones pràctiques per Python de Google

Aquesta guia de bones practiques de Python redactada per Google es basa en la guia oficial de Python PEP8. Qualsevol norma que no aparegui a continuació no és que no existeixi, sinó que prevaleix la redactada a PEP8 (veure Annex 4)

1. Regles d’estil generals de Python

1.1 Punt i coma

No acabar les línies amb punt i coma. No utilitzar punt i coma per posar dues declaracions a la mateixa línia.

1.2 Longitud de la línia

La longitud màxima de la línia és de màxim 80 caràcters.

1.3 Parèntesis

Utilitzar els parèntesis amb moderació.

Està bé, tot i que no és necessari, utilitzar parèntesis al voltant de tuples. No utilitzar-los en opcions de retorn o condicionals tret que s’utilitzin parèntesis per a la continuació implícita de la línia o per indicar una tupla.

1.4 Sagnia

Sagnar els blocs de codi amb 4 espais.

Mai utilitzar tabuladors o una barreja de tabulacions i espais. En els casos de continuació implícita de línia, cal alinear els elements del bloc de codi ja sigui verticalment o utilitzant una sagnia de 4 espais. En aquest cas no hi hauria d’haver res després del parèntesi obert o claudàtor a la primera línia.

1.5 Línies en blanc

- Dues línies en blanc entre definicions de nivell superior, ja siguin definicions de funció o de classe.
- Una línia en blanc entre les definicions del mètode, i entre la classe ("class") i el primer mètode.
- No hi ha d'haver cap línia en blanc després d'un "def".
- Utilitzar línies en blanc úniques segons el criteri apropiat dins de funcions o mètodes.
- Les línies en blanc no han d'estar ancorades a la definició.

1.6 Espais en blanc

- Seguir les regles tipogràfiques estàndard per a l'ús d'espais al voltant de la puntuació.
- No hi ha d'haver espais en blanc entre parèntesis o claudàtors.
- No hi ha d'haver espais en blanc ni abans ni després de posar una coma, punt o punt i coma. Exceptuant el final de línia de codi.
- No posar espais entre :, #, =, etc. per alinear codi
- Posar sempre espais a banda i banda entre aquests operadors binaris: assignació (=), assignació augmentada (+=, -= etc.), comparacions (==, <, >, !=, <>, <=, >=, in, not in, is, is not), booleans (and, or, not).

1.7 Línia Shebang

La majoria d'arxius .py no necessiten començar amb una línia #!. Iniciar el fitxer principal d'un programa amb #!/usr/bin/env python3 (per donar suport a virtualenvs) o #!/usr/bin/python3 per PEP-394[18]. Aquesta línia és utilitzada pel kernel per trobar l'intèrpret de Python, però és ignorada per Python per importar mòduls. Només és necessària en un fitxer destinat a ser executat directament.

1.8 Comentaris i docstrings

Assegurar-se que s'utilitzarà l'estil adequat per a mòduls, funcions, docstrings i comentaris en línia.

1.9 Strings

Utilitzar una f-string, l'operador %, o el mètode "format" per formatar strings, fins i tot quan tots els paràmetres són strings.

1.10 Arxius, Sockets, i fonts d'estat similar

Tancar explícitament els fitxers i els sockets quan s'acabin d'utilitzar. Aquesta regla s'estén als recursos que utilitzen els sockets internament, com ara connexions de bases de dades, i també a altres recursos que s'han de tancar de manera similar.

1.11 Comentaris "TODO"

- Utilitzar els comentaris "TODO" per al codi que sigui temporal, una solució a curt termini o prou bona, però no perfecta.
- L'objectiu és tenir un format "TODO" coherent que es pugui cercar per saber com obtenir més detalls.
- Un "TODO" no és un compromís que la persona referenciada resoldrà el problema. Per tant, quan es crea, gairebé sempre és el nom de qui l'ha creat el que es dona.

[18] https://peps.python.org/pep-0394/

- Si el "TODO" descriu modificacions futures, assegurar-se d'incloure una data o un esdeveniment específic.

1.12 Format d'importacions

Les importacions han d'estar en línies separades; hi ha excepcions per a les importacions de "typing" i "collections.abc".

1.13 Declaracions

En general, només una declaració per línia.

1.14 Getters i Setters

Les funcions "Getter" i "Setter" (també anomenades "accessors" i "mutators") s'han d'utilitzar quan proporcionen un rol o comportament significatiu per obtenir o establir el valor d'una variable. En particular, s'han d'utilitzar quan obtenir o fixar la variable és complex o el cost és significatiu, ja sigui en aquell precís moment o en un futur raonable.

1.15 Noms

Els noms de les funcions, els noms de les variables i els noms de fitxer han de ser descriptius; cal evitar abreviatures. En particular, no utilitzar abreviatures ambigües o desconegudes per als lectors fora del projecte, i no abreviar suprimint lletres dins d'una paraula.

1.16 Funció main

A Python, pydoc, així com les proves unitàries, requereixen que els mòduls siguin importables. Si un fitxer està destinat a ser utilitzat com a executable, la seva funcionalitat principal haurà d'estar en una funció principal, i el codi sempre ha de comprovar "if __name__ == '__main__'" abans d'executar el programa principal, de manera que no s'executi quan s'importa el mòdul.

1.17 Longitud de la funció

Preferentment funcions petites i centrades al seu objectiu principal.

1.18 Anotacions de tipus

Hi ha normes generals com: només anotar "self" o "cls" si és necessari per a una informació adequada. Si no s'ha d'expressar cap altra variable o cap opció de retorn, utilitzar "Any". No s'han d'anotar totes les funcions d'un mòdul.

Seguir les normes explicades anteriorment sobre espais i sagnat.

ANNEX 6: Regles i directrius de MISRA C:2012

Descripció de les columnes de dreta a esquerra: llistat de regles i directrius, nivell de classificació d'aquestes, i descripció. Les regles i directrius es classifiquen entres nivells segons necessitat: Necessària, recomanada i obligatòria.

REGLES I DIRECTRIUS	CLASSIFICACIÓ	DESCRIPCIÓ
D.1.1	Necessària	Qualsevol comportament definit per la implementació del qual depengui la sortida del programa ha de ser documentat i entès
D.2.1	Necessària	Tots els fitxers d'origen s'han de complir sense errors de compilació
D.3.1	Necessària	Tot el codi s'ha de rastrejar als requisits documentats
D.4.1	Necessària	S'han de minimitzar els errors en temps d'execució
D.4.2	Recomanada	S'ha de documentar tot l'ús del llenguatge assemblador
D.4.3	Necessària	El llenguatge ha d'estar marcat i aïllat
D.4.4	Recomanada	Les seccions de codi no s'han de "comentar"
D.4.5	Recomanada	Els identificadors en el mateix espai de noms amb visibilitat superposada han de ser tipogràficament inequívocs
D.4.6	Recomanada	Els tipus de lletra que indiquen la mida i el signe del tipus de dades numèriques s'han d'utilitzar en lloc dels tipus numèrics bàsics
D.4.7	Necessària	Si una funció retorna informació d'error, es provarà aquesta informació d'error
D.4.8	Recomanada	Si un punter d'una estructura o unió mai es deixa de fer dins d'una unitat de traducció, la implementació de l'objecte s'ha d'ocultar
D.4.9	Recomanada	S'ha d'utilitzar preferentment una funció en lloc d'un macro quan aquest és intercanviable
D.4.10	Necessària	S'han de prendre precaucions per evitar que el contingut d'un fitxer de capçalera s'inclogui més d'una vegada
D.4.11	Necessària	S'ha de comprovar la validesa dels valors passats a les funcions de les llibreries
D.4.12	Necessària	No s'ha d'utilitzar l'assignació de memòria dinàmica
D.4.13	Recomanada	Les funcions dissenyades per proporcionar operacions sobre un recurs s'han d'anomenar en una seqüència adequada
D.4.14	Necessària	S'ha de comprovar la validesa dels valors rebuts de fonts externes
R.1.1	Necessària	El programa no ha de contenir violacions de la sintaxi C estàndard i les restriccions, i no ha d'excedir els límits de traducció de la implementació
R.1.2	Recomanada	No s'han d'utilitzar extensions d'idioma
R.1.3	Necessària	No hi ha d'haver ocurrència de comportaments no especificats o crítics
R.1.4	Necessària	No s'han d'utilitzar característiques lingüístiques emergents
R.2.1	Necessària	Un projecte no ha de contenir codi inabastable
R.2.2	Necessària	No hi ha d'haver codi mort
R.2.3	Recomanada	Un projecte no ha de contenir tipus ni classes no utilitzats
R.2.4	Recomanada	Un projecte no ha de contenir etiquetes no utilitzats
R.2.5	Recomanada	Un projecte no ha de contenir macros no utilitzats
R.2.6	Recomanada	Una funció no ha de contenir classificacions no usades
R.2.7	Recomanada	No hi ha d'haver paràmetres que no s'utilitzin a les funcions
R.3.1	Necessària	Les seqüències de caràcters /* i // no s'utilitzaran dins d'un comentari
R.3.2	Necessària	L'unió de línies no s'utilitzarà en // comentaris
R.4.1	Necessària	Les seqüències d'escapament octals i hexadecimals s'han d'acabar
R.4.2	Recomanada	No s'han d'utilitzar trígrafs
R.5.1	Necessària	Els identificadors externs s'han de distingir
R.5.2	Necessària	Els identificadors declarats en el mateix àmbit i espai de nom han de ser diferents
R.5.3	Necessària	Un identificador declarat en un àmbit interior no ha d'amagar un identificador declarat en un àmbit exterior
R.5.4	Necessària	Els identificadors de macros han de ser diferents
R.5.5	Necessària	Els identificadors han de ser diferents dels noms dels macros
R.5.6	Necessària	Una definició de classe i tipo ha de ser única
R.5.7	Necessària	Un nom d'etiqueta serà un identificador únic
R.5.8	Necessària	Els identificadors que defineixen objectes o funcions amb vinculació externa seran únics
R.5.9	Recomanada	Els identificadors que defineixen objectes o funcions amb vinculació interna han de ser únics
R.6.1	Necessària	L'estructura de bits només s'ha de declarar amb un tipus o classe adequat
R.6.2	Necessària	Les estructures de bits amb nom d'un sol bit no han de ser signats
R.7.1	Necessària	No s'han d'utilitzar constants octals
R.7.2	Necessària	El sufix "u" o "U" s'ha d'utilitzar en constants enteres representades per un número no signat
R.7.3	Necessària	El caràcter en minúscula 'l' no s'utilitzarà en un sufix literal
R.7.4	Necessària	Un string no ha d'estar assignat a un objecte a no ser que l'objecte estigui apuntat a un caràcter qualificat.
R.8.1	Necessària	El tipus s'ha d'especificar explícitament
R.8.2	Necessària	Les funcions han de tenir un prototip amb paràmetres especificats
R.8.3	Necessària	Totes les declaracions d'un objecte o funció han d'utilitzar els mateixos noms i tipus de qualificadors

REGLES I DIRECTRIUS	CLASSIFICACIÓ	DESCRIPCIÓ
R.8.4	Necessària	Una declaració compatible serà visible quan es defineixi un objecte o funció amb vinculació externa.
R.8.5	Necessària	Un objecte o funció externa es declararà una vegada en un i només un fitxer
R.8.6	Necessària	Un identificador amb enllaç extern ha de tenir exactament una definició externa
R.8.7	Recomanada	Les funcions i els objectes no s'han de definir amb enllaç extern si es fa referència en una sola unitat de traducció
R.8.8	Necessària	L'especificador de classe d'emmagatzematge estàtic s'ha d'utilitzar en totes les declaracions d'objectes i funcions que tinguin vinculació interna
R.8.9	Recomanada	Un objecte s'ha de definir a l'àmbit de bloc si el seu identificador només apareix en una sola funció
R.8.10	Necessària	Una funció en línia s'ha de declarar amb la classe d'emmagatzematge estàtic
R.8.11	Recomanada	Quan es declara una matriu amb enllaç extern, s'ha d'especificar explícitament la seva mida
R.8.12	Necessària	Dins d'una llista d'enumeradors, el valor d'una constant d'enumeració especificada implícitament serà únic
R.8.13	Recomanada	Un punter ha d'apuntar a un tipus qualificat sempre que sigui possible
R.8.14	Necessària	No s'ha d'utilitzar el qualificador de tipus de restricció
R.9.1	Obligatòria	El valor d'un objecte amb durada d'emmagatzematge automàtic no s'ha de llegir abans d'haver-lo definit
R.9.2	Necessària	L'inicialitzador d'un agregat o unió s'ha de tancar entre parèntesis clau { }
R.9.3	Necessària	Les matrius no s'han d'inicialitzar parcialment
R.9.4	Necessària	Un element d'un objecte no s'ha d'inicialitzar més d'una vegada
R.9.5	Necessària	Quan s'utilitzin inicialitzadors designats per inicialitzar un objecte d'una matriu, la mida de la matriu s'ha d'especificar explícitament
R.10.1	Necessària	Els operands no han de ser d'un tipus essencial inadequat
R.10.2	Necessària	Les expressions de tipus de caràcter essencial no s'han d'utilitzar de manera inadequada en les operacions d'addició
R.10.3	Necessària	El valor d'una expressió no s'ha d'assignar a un objecte amb un tipus essencial més limitat o d'una categoria de tipus essencial diferent
R.10.4	Necessària	Tots dos operands d'un operador en què es realitzin les conversions aritmètiques habituals tindran la mateixa categoria de tipus essencial
R.10.5	Recomanada	El valor d'una expressió no s'ha d'emetre a un tipus essencial inadequat
R.10.6	Necessària	El valor d'una expressió composta no s'ha d'assignar a un objecte amb un tipus essencial més ampli
R.10.7	Necessària	Si una expressió composta està sent utilitzada com a operand d'un operador on les conversions aritmètiques es tornen a realitzar, l'altre operand no tindrà un tipus essencial més ampli
R.10.8	Necessària	El valor d'una expressió composta no s'ha de llançar a una categoria de tipus essencial diferent o a un tipus essencial més ampli
R.11.1	Necessària	No s'han de realitzar conversions entre un punter d'una funció ni cap altre tipus
R.11.2	Necessària	No s'han de realitzar conversions entre un punter incomplet i qualsevol altre tipus
R.11.3	Necessària	No s'han de realitzar conversions entre un punter d'un tipus d'objecte i un punter a un tipus d'objecte diferent
R.11.4	Recomanada	No s'ha de realitzar una conversió entre un punter i un objecte i un enter

REGLES I DIRECTRIUS	CLASSIFICACIÓ	DESCRIPCIÓ
R.11.5	Recomanada	No s'ha de realitzar una conversió entre un punter al buit i un punter a un objecte
R.11.6	Necessària	No s'ha de realitzar una conversió entre un punter al buit i un tipus aritmètic
R.11.7	Necessària	No s'ha de realitzar una conversió entre un punter d'un objecte i un tipus aritmètic no enter
R.11.8	Necessària	Un estil no ha d'eliminar cap correctiu de constants o qualificació volàtil del tipus punter per una variable punter
R.11.9	Necessària	El macro NULL ha d'estar només permès com a enter nul punter constant
R.12.1	Recomanada	La precedència dels operadors dins de les expressions s'ha de fer explícita
R.12.2	Necessària	L'operand dret d'un operador de desplaçament es troba en el rang de zero a un menys que l'amplada en bits de l'essencial
R.12.3	Recomanada	No s'ha d'utilitzar l'operador de comes
R.12.4	Recomanada	L'avaluació de les expressions constants no ha de conduir a un ajustament enter no signat
R.12.5	Obligatòria	L'operador no ha de tenir un operand que sigui una funció paràmetre declarada com "array of type"
R.13.1	Necessària	Les llistes inicialitzadores no han de contenir efectes secundaris persistents
R.13.2	Necessària	El valor d'una expressió i els seus efectes secundaris persistents seran els mateixos en totes les ordres d'avaluació permeses.
R.13.3	Recomanada	Una expressió completa que contingui un operador d'increment (++) o decrement (- -) no hauria de tenir altres efectes secundaris potencials que el causat per l'operador d'increment o decrements.
R.13.4	Recomanada	No s'ha d'utilitzar el resultat d'un operador d'assignació
R.13.5	Necessària	L'operand de la dreta d'un operador lògic && o || no ha de contenir efectes secundaris persistents
R.13.6	Obligatòria	L'operand de l'operador sizeof no ha de contenir cap expressió que tingui efectes secundaris potencials
R.14.1	Necessària	Un loop comptador no ha de tenir tipus flotants (float numbers)
R.14.2	Necessària	Un loop "for" ha d'estar ben format
R.14.3	Necessària	Expressions de control no ha de ser invariant
R.14.4	Necessària	L'expressió de control d'una declaració "if" i l'expressió controladora d'una declaració d'iteració han de ser essencialment de tipus booleà.
R.15.1	Recomanada	No s'ha d'utilitzar la declaració goto
R.15.2	Necessària	La declaració goto ha de saltar a una etiqueta declarada posteriorment en la mateixa funció
R.15.3	Necessària	Qualsevol etiqueta a la qual es faci referència en una declaració goto serà declarada en el mateix bloc, o en qualsevol bloc que adjunti la declaració goto.
R.15.4	Recomanada	No s'ha d'utilitzar més d'una sentència de ruptura o goto per finalitzar qualsevol sentència d'iteració
R.15.5	Recomanada	Una funció ha de tenir un únic punt de sortida al final
R.15.6	Necessària	El cos d'una declaració d'iteració o d'una declaració de selecció ha de ser una declaració composta
R.15.7	Necessària	Totes les construccions if. . else s'han de finalitzar amb una altra declaració
R.16.1	Necessària	Totes les declaracions de commutació han d'estar ben formades

REGLES I DIRECTRIUS	CLASSIFICACIÓ	DESCRIPCIÓ
R.16.2	Necessària	Una etiqueta switch només s'ha d'utilitzar quan la declaració composta més propera és el cos d'una sentència switch
R.16.3	Necessària	Una declaració de ruptura incondicional ha de finalitzar cada clàusula de commutació
R.16.4	Necessària	Cada sentència del commutador ha de tenir una etiqueta per defecte
R.16.5	Necessària	Una etiqueta per defecte ha d'aparèixer com a primera o última etiqueta de commutació d'una sentència de commutació
R.16.6	Necessària	Cada sentència de commutació ha de tenir almenys dues clàusules de commutació
R.16.7	Necessària	Una expressió de commutador no tindrà essencialment tipus booleà
R.17.1	Necessària	Les característiques <stdarg.h> no s'han d'utilitzar
R.17.2	Necessària	Les funcions no s'han d'anomenar a si mateixes, ni directament ni indirectament.
R.17.3	Obligatòria	Una funció no s'ha de declarar implícitament
R.17.4	Obligatòria	Tots els camins de sortida d'una funció amb tipus de retorn no nul tindran una declaració de retorn explícita amb una expressió
R.17.5	Recomanada	L'argument de funció corresponent a un paràmetre declarat que té un tipus de matriu tindrà un nombre adequat d'elements
R.17.6	Obligatòria	La declaració d'un paràmetre d'una matriu no ha de contenir la paraula clau estàtica entre els [ ]
R.17.7	Necessària	S'utilitzarà el valor retornat per una funció que tingui un tipus de retorn no nul
R.17.8	Recomanada	No s'ha de modificar un paràmetre de funció
R.18.1	Necessària	Un punter resultant d'aritmètica en un punter operand ha d'adreçar un element de la mateixa matriu del punter operand.
R.18.2	Necessària	La resta entre punters només s'ha d'aplicar a punters que s'adrecen a elements de la mateixa matriu.
R.18.3	Necessària	Els operadors relacionals >, >=, < i <= no s'han d'aplicar a objectes de tipus punter, excepte quan apuntin al mateix objecte.
R.18.4	Recomanada	Els operadors +, -, +=, -= no s'han d'aplicar a una expressió de tipus de punter
R.18.5	Recomanada	Les declaracions no han de contenir més de dos nivells de nidificació de punter
R.18.6	Necessària	L'adreça d'un objecte amb emmagatzematge automàtic no s'ha de copiar a un altre objecte que persisteixi després que el primer objecte hagi deixat d'existir.
R.18.7	Necessària	Els membres d'una matriu flexible no s'han de declarar
R.18.8	Necessària	El tipus de matrius variables en llargada no s'han d'utilitzar
R.19.1	Obligatòria	Un objecte no s'ha d'assignar ni copiar a un objecte superposat
R.19.2	Recomanada	La unió per paraules clau no s'ha d'utilitzar
R.20.1	Recomanada	#include només pot anar precedit de preprocessadors directrius o comentaris
R.20.2	Necessària	Els caràcters "," o \ i /* o // no han d'anar d'estar en la capçalera del nom del fitxer
R.20.3	Necessària	La directriu #include ha d'anar seguida de la seqüència "filename" o <filename>
R.20.4	Necessària	Una macro no es defineix amb el mateix nom que una paraula clau

REGLES I DIRECTRIUS	CLASSIFICACIÓ	DESCRIPCIÓ
R.20.5	Recomanada	#undef no s'ha d'utilitzar
R.20.6	Necessària	Els tokens que semblen una directiva de preprocessament no es produiran dins d'un argument macro
R.20.7	Necessària	Les expressions resultants de l'expansió dels paràmetres macro s'han d'incloure entre parèntesis
R.20.8	Necessària	L'expressió de control d'una directiva de preprocessament #if o #elif s'avaluarà a 0 o 1
R.20.9	Necessària	Tots els identificadors utilitzats en l'expressió de control de les directives de preprocessament #if o #elif s'han de definir abans de l'avaluació.
R.20.10	Recomanada	Els operadors de processador # i ## no s'han d'utilitzar
R.20.11	Necessària	Un paràmetre macro seguit immediatament per un operador "#"no pot anar seguit immediatament per un operador "##"
R.20.12	Necessària	Un paràmetre macro utilitzat com a operand als operadors # o ##, que està subjecte a un nou reemplaçament macro, només s'utilitzarà com a operand per a aquests operadors
R.20.13	Necessària	Una línia el primer testimoni de la qual sigui # serà una directiva de preprocessament vàlida
R.20.14	Necessària	Tots els #else, #elif i #endif de processadors directrius han de residir en el mateix fitxer que #if, #ifdef o #ifndef al qual pertanyen
R.21.1	Necessària	#define i #undef no s'han d'utilitzar en un identificador reservat o en un macro nom reservat
R.21.2	Necessària	No es declararà un identificador reservat ni un nom de macro reservat
R.21.3	Necessària	La distribució de memòria de les funcions <stdlib.h> no s'han d'utilitzar
R.21.4	Necessària	No s'utilitzarà el fitxer de capçalera estàndard <setjmp.h>
R.21.5	Necessària	No s'utilitzarà el fitxer de capçalera estàndard <signal.h>
R.21.6	Necessària	No s'utilitzaran les rutines d'entrada/sortida de la llibreria estàndard
R.21.7	Necessària	No s'utilitzaran les funcions estàndard de la Llibreria atof, atoi, atol i atol
R.21.8	Necessària	No s'utilitzaran les funcions de terminació estàndard de la llibreria
R.21.9	Necessària	Les funcions de llibreria estàndard bsearch i qsort de <stlib.h> no s'utilitzaran
R.21.10	Necessària	No s'utilitzaran les rutines estàndard d'hora i data de la llibreria
R.21.11	Necessària	No s'utilitzarà el fitxer de capçalera estàndard <tgmath.h>
R.21.12	Recomanada	Les característiques de gestió d'excepcions <fenv.h> no s'ha d'utilitzar
R.21.13	Obligatòria	Qualsevol valor passat a una funció <ctype.h> serà representable com un caràcter sense signar o valor EOF
R.21.14	Necessària	El memcmp de la funció llibreria estàndard no s'utilitzarà per comparar cadenes acabades nul·les
R.21.15	Necessària	Els arguments del punter a les funcions de la llibreria estàndard memcpy, memmove i memcmp seran punters de versions qualificades o no qualificades de tipus compatibles
R.21.16	Necessària	Els arguments de la funció memcmp de la llibreria estàndard han d'apuntar a un tipus de punter, tipus assignat, boolean, o enum.
R.21.17	Obligatòria	L'ús de les funcions de maneig de cadenes "from" no donarà lloc a accessos més enllà dels límits dels objectes referenciats pels seus paràmetres apuntadors
R.21.18	Obligatòria	L'argument size_t passat a qualsevol funció <string.h> ha tenir un valor apropiat

REGLES I DIRECTRIUS	CLASSIFICACIÓ	DESCRIPCIÓ
R.21.19	Obligatòria	Els punters retornats per les funcions de la llibreria estàndard localeconv, getenv, setlocale o, strerror només s'utilitzaran com si tinguessin punter a tipus qualificat const
R.21.20	Obligatòria	El valor retornat per una execució d'una de les funcions de la llibreria Standard asctime, ctime, gmtime, localtime, localeconv, getenv, setlocale o strerro, no s'ha d'utilitzar després d'una execució subseqüent a la mateixa funció
R.21.21	Necessària	Les funcions de sistema <stdlib.h> de llibreries estàndards no s'han d'uti-litzar
R.22.1	Necessària	Tots els recursos obtinguts dinàmicament mitjançant funcions de llibreria estàndard s'han d'alliberar explícitament
R.22.2	Obligatòria	Un bloc de memòria només s'alliberarà si s'assigna mitjançant una funció de llibreria estàndard
R.22.3	Necessària	El mateix fitxer no estarà obert per a l'accés de lectura i escriptura al mateix temps en diferents fluxos
R.22.4	Obligatòria	No hi haurà cap intent d'escriure a un flux que s'ha obert només per a lectura
R.22.5	Obligatòria	Un punter cap a un objecte FILE no pot estar des-referenciat
R.22.6	Obligatòria	El valor punter d'un FILE no es pot utilitzar fins que el stream associat estigui tancat.
R.22.7	Necessària	La macro EOF només s'ha de comparar amb el valor de retorn no modificat de qualsevol funció de llibreria estàndard capaç de retornar EOF
R.22.8	Necessària	El valor errno ha de ser 0 abans de executar la funció errno-setting-function
R.22.9	Necessària	El valor errno ha de ser testejar contra el valor 0 després d'executar la funció errno-setting-function
R.22.10	Necessària	El valor errno només pot ser testejat quan la última funció a ser executada ha estat la funció errno-setting-function

TIC
Salut Social

Programa Salut/ IA
Programa Salut/ IA

Generalitat
de Catalunya