# Integrita dat, bezpečnost, logování, kontrola vstupů, zpracování chyb

## Integrita dat a Bezpečnost

Integrita dat zajišťuje, aby data zůstala konzistentní a nedocházelo k neoprávněným změnám či poškození dat během zpracování nebo přenosu. Integrita dat je důležitá pro zajistění správného fungování aplikací a zabraňuje případnému poškození či ztrátě dat.

Integrita:

* Kontrola datových typů – Kontrola datových typů zabraňuje neplatným datovým typům a formátům. Pokud je předpokládána určitá hodnota datového typu a není správně zadána, může dojít k nesprávným výsledkům a narušení integrity dat.
* Validace vstupů – Před zpracováním vstupních dat by měla být provedena validace, aby se zabránilo vstupu neplatných nebo nebezpečných dat. Validace může zahrnovat kontrolu datového typu, formátování a kontroly na případné nebezpečné prvky, jako jsou SQL injection nebo cross-site scripting.
* Zálohování dat – Zálohování dat je důležité pro zajištění integrity dat v případě, že dojde k havárii nebo ztrátě dat. V Pythonu existuje několik knihoven pro zálohování dat, jako je například knihovna tarfile.

Bezpečnost v Pythonu se týká ochrany aplikací před různými druhy hrozeb a zabezpečení dat před neoprávněným přístupem.

Bezpečnost:

* Kontrola datových typů – Kontrola datových typů zabraňuje neplatným datovým typům a formátům. Pokud je předpokládána určitá hodnota datového typu a není správně zadána, může dojít k nesprávným výsledkům a narušení integrity dat.
* Šifrování dat – Šifrování dat pomáhá chránit citlivá data před neoprávněným přístupem. V Pythonu jsou k dispozici různé knihovny pro šifrování dat, jako jsou například knihovny Crypto nebo bcrypt.
* Ověřování identity – Při práci s citlivými daty je důležité ověřit, že osoba nebo aplikace, která data požaduje, je oprávněna k přístupu. K ověřování identity se obvykle používají různé metody, jako jsou například hesla, tokeny nebo biometrické údaje.
* Zabezpečení proti útokům – Pythonové aplikace by měly být chráněny proti různým typům útoků, jako jsou například útoky na přístupová hesla, útoky typu denial-of-service nebo útoky typu man-in-the-middle. K zabezpečení proti těmto útokům může být použita různá opatření, jako jsou například bezpečné protokoly komunikace, řízení přístupu nebo monitorování aktivit
* Aktualizace a správa softwaru – Aktualizace softwaru je důležitá pro zajištění bezpečnosti a stability aplikace. Při vývoji Pythonové aplikace je tedy důležité pravidelně aktualizovat používané knihovny a spravovat celkovou infrastrukturu aplikace.

Bráníme se proti:

* Útoky typu SQL injection - SQL injection je útok, který využívá chyb v aplikaci a umožňuje útočníkovi manipulovat s databází. Aby se zabránilo útoku typu SQL injection, je nutné používat parametrizované dotazy a escapovat speciální znaky.
* Cross-site scripting (XSS) - XSS je útok, který umožňuje útočníkovi vkládat do stránky škodlivý kód, který může ovlivnit chování stránky. Aby se zabránilo útoku typu XSS, je nutné escapovat uživatelský vstup a používat bezpečné metody pro manipulaci s HTML.
* Útoky na hesla - Útoky na hesla jsou útoky, při kterých útočník snaží získat přístup k aplikaci pomocí neplatného uživatelského jména a hesla. Aby se zabránilo útokům na hesla, je nutné používat silná hesla, hashování hesel a dvoufaktorovou autentizaci.
* Útoky typu denial-of-service (DoS) - DoS útoky jsou útoky, které mají za cíl zpomalit nebo zastavit výkon aplikace. Aby se zabránilo útokům typu DoS, je nutné používat bezpečná konfigurační nastavení a zabezpečit serverovou infrastrukturu.
* Neoprávněný přístup k datům - Neoprávněný přístup k datům může být způsoben nedostatečnou ochranou přístupových práv nebo nebezpečnými konfiguracemi. Aby se zabránilo neoprávněnému přístupu k datům, je nutné používat přesná oprávnění a bezpečné metody pro manipulaci s daty.

## Logování

Logování je akt udržování informací o stavech v programu jako jsou: problémy, erorry nebo jenom informace o aktuální operaci. Logování je podstatné pro pochopení chování programu a pro jednodušší debuggování neočekávaných problémů.

Logování v Pythonu se provádí pomocí modulu logging modulu, který umožňuje vytvářet a ukládat logy zpráv různého typu, jako jsou například chyby, varování, informace a tak dále.

**import** **logging**

**logging**.basicConfig(format='%(asctime)s %(levelname)s: %(message)s', datefmt='%Y-%m-%d %H:%M:%S',filename='example.log', level=logging.DEBUG)

logger = **logging**.getLogger(\_\_name\_\_)#vytvoreni instance

**logging**.debug('Toto je debugovací zpráva')

**logging**.info('Toto je informační zpráva')

**logging**.warning('Toto je varování')

**logging**.error('Toto je chyba')

**logging**.critical('Toto je kritická chyba')

## Kontrola vstupů

Vstupy uživatelů mohou být různorodé, a ne vždy se jedná o platná data, která je možné použít v aplikaci. Proto je důležité provádět kontrolu vstupů v Pythonu.

Co můžeme například kontrolovat u vstupních dat:

* Kontrola datového typu
* Kontrola rozsahu hodnot
* Kontrola délky řetězců
* Kontrola platnosti dat (například u datumů)

Kontrolu můžeme provádět pomocí:

* Podmínky (if/else) - Podmínky umožňují ověřit, zda splňují určité požadavky, a následně provést určitou akci.

number = 5

**if** number > 0:

print("Číslo je kladné.")

**else**:

print("Číslo je záporné nebo nula.")

* Aserce (assert) - Aserce je funkce, která kontroluje, zda je podmínka pravdivá, a v případě, že je podmínka nepravdivá, vyvolá výjimku.

age = 25

**assert** age > 0, "Věk musí být kladný."

* Regulární výrazy (regex) - Regulární výrazy umožňují definovat složitější podmínky pro ověřování dat. Regulární výrazy lze použít k ověřování formátu textových řetězců.

import re

email = "johndoe@example.com"

**if** re.match(r"[^@]+@[^@]+\.[^@]+", email):

print("Emailová adresa je platná.")

**else**:

print("Emailová adresa není platná.")

* Try/except bloky - Try/except bloky umožňují zachytit výjimky a provést určitou akci v případě, že k výjimce došlo.

**try**:

number = int(input("Zadejte číslo: "))

**except**:

print("Zadali jste neplatné číslo.")

## Zpracování výjimek

Zpracování chyb je proces, kterým se zabýváme, když se v programu vyskytne výjimka. Při použití výjimek v Pythonu může programátor určit, co se má stát, pokud dojde k výjimce. Používá se blok try-except, který odchytne výjimku a provede nějaký kód, pokud k ní dojde. Používáme kód pro zpracování výjimek k odchycení chyb a provedení určitých kroků, aby se program nezastavil a nepřestal fungovat.

Následující příklad ukazuje, jak použít blok try-except, final a else k odchycení výjimky a provedení alternativního kódu:

**try**:

result = x / y

**except** **ZeroDivisionError**:

print("division by zero!")#vyskoci chyba

**else**:

print("result is", result)#nevyskoci chyba

**finally**:

print("executing finally clause")#probehne vzdy

pokud chceme využít více except bloků s nebo bez defaultní expect blok můžeme učinit takto (na co nesmíme zapomenout je, že defaultní except blok bez parametrů jde vždy až jako poslední)

**try**:

f = open('myfile.txt')

s = f.readline()

i = int(s.strip())

**except** **OSError** **as** err:

print("OS error:", err)

**except** **ValueError**:

print("Could not convert data to an integer.")

**except** **Exception** **as** err:

print(f"Unexpected **{**err**=}**, **{**type(err)**=}**")

**except**:

print(f"Some other error")

### Exception Group

**def** f():

**...**  excs = [**OSError**('error 1'), **SystemError**('error 2')]

**...**  **raise** ExceptionGroup('there were problems', excs)

f()

+ Exception Group Traceback (most recent call last):

| File "<stdin>", line 1, in <module>

| File "<stdin>", line 3, in f

| ExceptionGroup: there were problems

+-+---------------- 1 ----------------

| OSError: error 1

+---------------- 2 ----------------

| SystemError: error 2

+------------------------------------

### Custom Exception:

**class** SalaryNotInRangeError(Exception):

    """Exception raised for errors in the input salary.

    Attributes:

        salary -- input salary which caused the error

        message -- explanation of the error

    """

**def** \_\_init\_\_(self, salary, message="Salary is not in (5000, 15000) range"):

        self.salary = salary

        self.message = message

        super().\_\_init\_\_(self.message)

salary = int(input("Enter salary amount: "))

if not 5000 < salary < 15000:

    raise SalaryNotInRangeError(salary)

V Pythonu je třída BaseException základní třídou pro všechny výjimky, od kterých jsou ostatní výjimky odvozeny. Tato třída definuje několik základních atributů, které jsou důležité pro správné zacházení s výjimkami. Zde jsou tyto atributy:

* args: Tupple obsahující argumenty výjimky. Tyto argumenty jsou předány konstruktoru výjimky a lze je použít k zobrazení detailů o chybě.
* with\_traceback(tb): Metoda, která přiřadí zásobník volání traceback k výjimce. Pokud není poskytnuto žádné argument, použije se aktuální traceback.
* \_\_cause\_\_: Odkazuje na další výjimku, která způsobila tuto výjimku. Pokud tato výjimka není důsledkem jiné výjimky, bude hodnota tohoto atributu None.
* \_\_context\_\_: Odkazuje na kontext této výjimky, například chybu, která ji předcházela. Pokud tato výjimka není v kontextu žádné jiné výjimky, bude hodnota tohoto atributu None.
* \_\_traceback\_\_: Zásobník volání traceback, který byl přiřazen pomocí metody with\_traceback(). Pokud nebyla použita metoda with\_traceback(), bude hodnota tohoto atributu None.