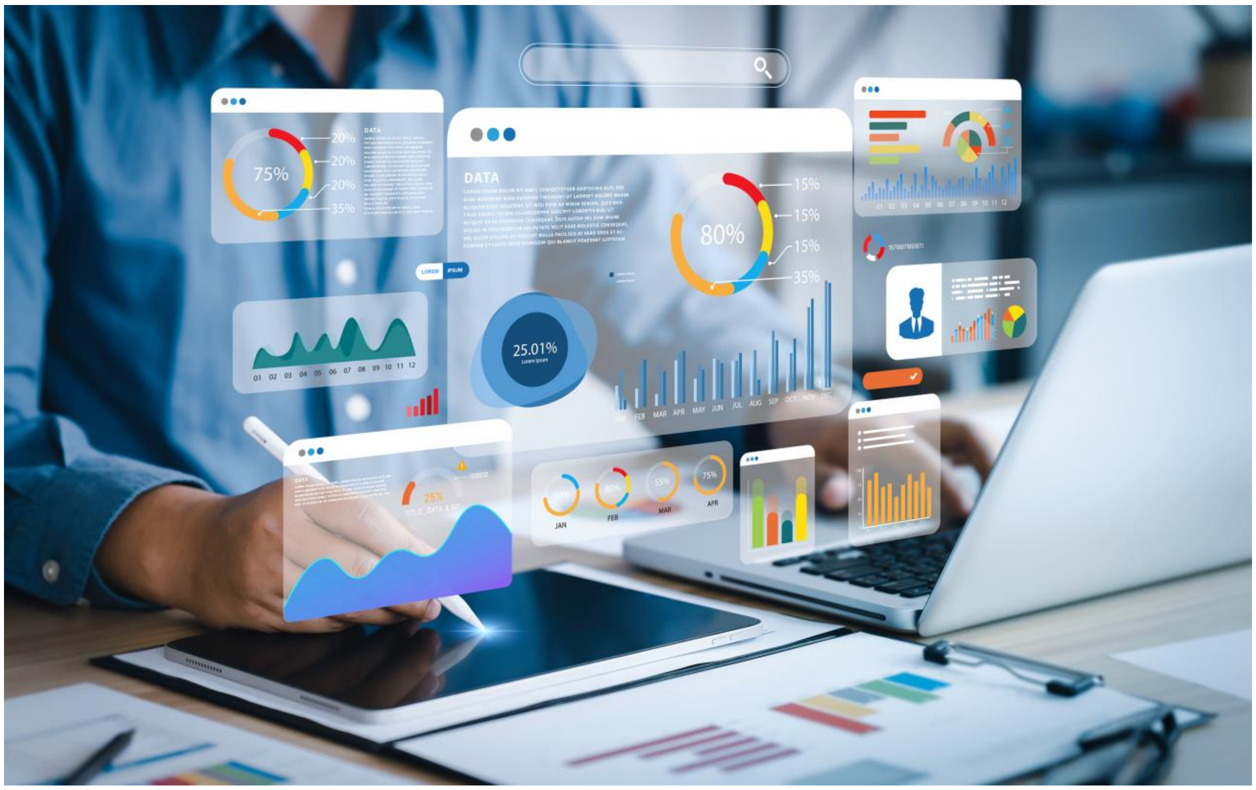
ACC 424 Accounting Information System

Automating internal controls testing
Python notes for 6.4

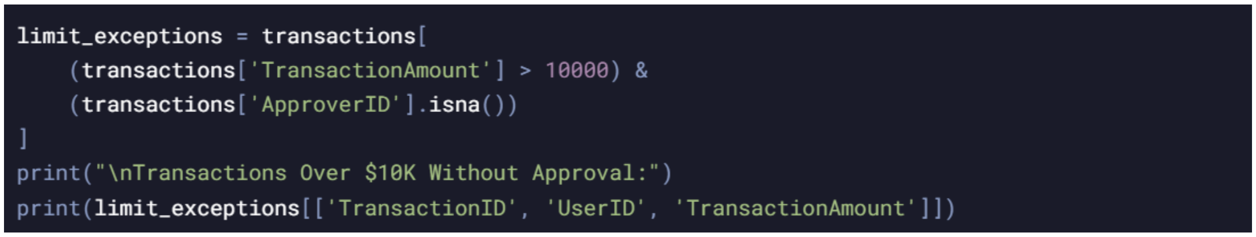


Section 001 MW 9:30 AM – 10:45 AM at Rm. 257 over Jan 13 – May 07
Section 002 MW 11:00 AM – 12:15 AM at Rm. 127 over Jan 13 – May 07

6.4. Hands-on automating internal controls testing with Python (Continued)

A. Transaction Limit Check

A Transaction Limit Check is critical internal control ensuring that no single transaction exceeds **predefined** monetary thresholds without proper **approval**. So, high-value transactions receive appropriate **oversight**.



```
limit_exceptions = transactions[
    (transactions['TransactionAmount'] > 10000) &
    (transactions['ApproverID'].isna())
]
print("\nTransactions Over $10K Without Approval:")
print(limit_exceptions[['TransactionID', 'UserID', 'TransactionAmount']])
```

1. **transactions['TransactionAmount'] > 10000**
   - This part of the code evaluates whether each transaction's amount exceeds the $10,000 threshold. When Python executes this comparison, it goes through every row in the 'TransactionAmount' column and checks if the value is greater than 10,000. The result is a **Boolean** series (True/False values) where True indicates transactions that **violate** the amount threshold.
   - This is the "what" we're checking. This represents the quantitative aspect of the control – we're identifying which transactions are large enough to require special approval **scrutiny**.
2. **transactions['ApproverID'].isna()**
   - This portion examines the approval status of each transaction. The .isna() method checks for **missing** values (NaN) in the 'ApproverID' column, which would indicate that **no** approver was assigned to that transaction. Like the amount check,

this generates a Boolean series where **True** means the transaction lacks proper approval.

   - This verifies the "who" - whether proper **approval** exists. This addresses the procedural aspect of the control - verifying that proper **oversight** was exercised for significant transactions.

3. **& (Logical AND)**
   - The **operator** symbol "&" combines the above two conditions using Boolean logic. Importantly, it performs an element-wise **comparison** between the two Boolean series. Only transactions where **BOTH** conditions are True (amount exceeds limit AND approver is missing) will be included in the final result.
   - This creates **complete** control condition that satisfies our internal control requirement.

Output Interpretation

For a transaction to appear in limit_exceptions, it must:

1. Have an amount > $10,000
2. Have no approver listed (ApproverID is blank)

When these components work together, the code creates a filtered DataFrame (limit_exceptions) that contains only the **problematic** transactions. For a transaction to appear in this output, it must simultaneously satisfy two conditions: First, its monetary value must cross our $10,000 threshold, indicating it's **significant** enough to require oversight. Second, it must **lack** an approver ID, meaning the required approval step was either skipped or not properly recorded. The combination is what makes this control effective - we're not just looking for large transactions or missing approvals in isolation, but specifically for **large** transactions that didn't receive the required **approval**.

B. Segregation of Duties (SoD) Check

Segregation of Duties is a key internal control designed to prevent fraud and errors. The basic idea is simple: **no single individual** should control all aspects of a critical financial transaction. Duties should be **divided** among different people to reduce the risk of intentional wrongdoing. SoD check identifies **violations** where the **same** user both initiated and approved a transaction, which violates this fundamental internal control principle.

```
# Merge transactions with activity logs to find conflicts
sod_violations = pd.merge(
    transactions,
    activity_log,
    on='UserID',
    how='inner'
)
sod_violations = sod_violations[
    sod_violations['Action'].str.contains('Approve')
]
print("\nSoD Violations (User Initiated & Approved):")
print(sod_violations[['TransactionID', 'UserID', 'Action']])
```

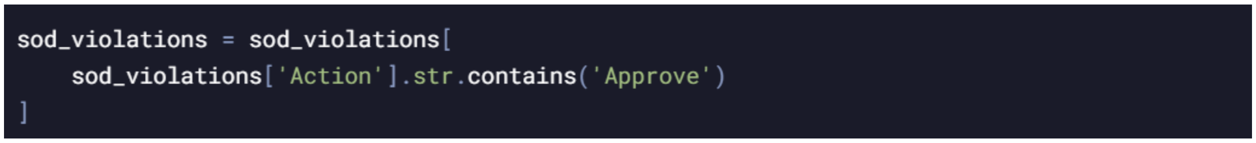
Data Merging:

```
sod_violations = pd.merge(
    transactions,
    activity_log,
    on='UserID',
    how='inner'
)
```

- The code performs a database-style **join** between two datasets:

    - transactions DataFrame: Contains all financial transaction records with initiator (UserID), i.e., who initiated transactions.
    - activity_log DataFrame: Contains a record of every user action in the system including approvals.
- It joins them using UserID as the key (on='UserID'). By merging these on UserID, we’re essentially creating a **combined** dataset that shows all instances where the **same user** appears in both the transaction initiator field and the activity log.
- how='inner' ensures only matching users in both datasets are kept. The 'inner' join type means we only keep records where the UserID exists in **both** datasets - these are our potential violations that need investigation.

Approval Action Filter:



```
sod_violations = sod_violations[
    sod_violations['Action'].str.contains('Approve')
]
```

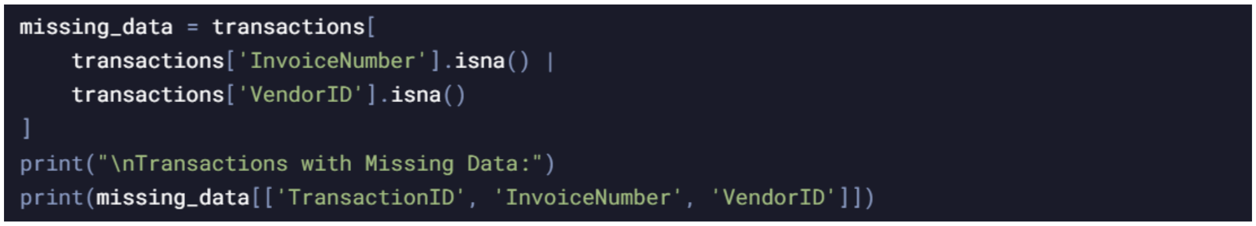
- Filters the merged data to only show records where:
    - The same UserID appears in both datasets (**initiator** and **approver**).
    - The str.contains('Approve') method scans the Action column for any **string** entries that include the word "Approve" (like "Approve Payment" or "Approve Transaction"). The activity log might contain many types of user actions (creating, editing, viewing), but we only care about **approvals** for this control. This filtering ensures we don’t flag cases where a user merely viewed or edited their own transaction, which wouldn’t necessarily violate **SoD** principles.

- This catches cases where a user **approved** their own **initiated** transactions. The fundamental control principle here is that **no** single individual should have the ability to both **initiate** and **approve** the same transaction. This separation prevents employees from creating fraudulent transactions and then approving them themselves.

C. Data Validation Check

Data validation performs a critical **completeness** check on transaction records by verifying that essential fields contain valid data. The validation focuses on two key fields InvoiceNumber and VendorID. **InvoiceNumber** field maintains proper audit trails, as it links payments to their corresponding source documents. **VendorID** field is critical for accurate vendor management and year-end reporting.

Data validation operates by systematically scanning **all** transactions to identify records where either of these mandatory fields contain null or missing values. We ensure that all transactions maintain complete supporting documentation and that vendor relationships are properly recorded.



```
missing_data = transactions[
    transactions['InvoiceNumber'].isna() |
    transactions['VendorID'].isna()
]
print("\nTransactions with Missing Data:")
print(missing_data[['TransactionID', 'InvoiceNumber', 'VendorID']])
```

1. **Boolean Mask Creation**:
   - The isna() method generates a True/False mask indicating **missing** values.
   - Applied separately to both InvoiceNumber and VendorID columns. When we apply transactions['InvoiceNumber'].isna(), pandas internally scans the entire

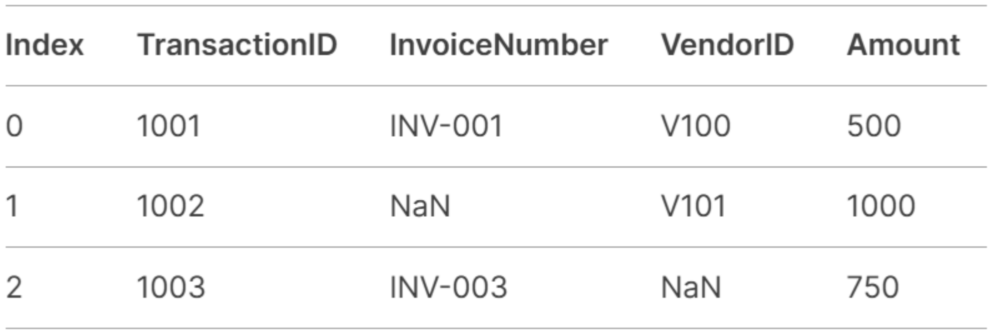
InvoiceNumber column and construct an array of Boolean values (True/False) of identical length to the DataFrame. Each True value represents a cell where the InvoiceNumber is **missing** (NaN), while False indicates **valid** entries. This same process occurs simultaneously for the VendorID column, creating a second parallel Boolean array.

- Creates two parallel arrays of Boolean values matching the DataFrame's length.

2. **Logical OR Operation**:
   - The pipe operator | performs element-wise **OR** comparison.
   - For each row position, if either the InvoiceNumber mask shows True (missing) **OR** the VendorID mask shows True (missing), the combined mask will contain True for that row position.
   - That is, For each transaction, if **EITHER** the InvoiceNumber switch is on (missing) OR the VendorID switch is on (missing), then the whole transaction gets a **red flag**.

3. **DataFrame Filtering**:
   - The Boolean mask indexes the original DataFrame. Each True/False value corresponds exactly to a row in the original transactions DataFrame. When we write transactions[mask], pandas performs an **alignment** check first - verifying the mask length matches the DataFrame's row count and that their indices align properly. This ensures accurate row selection.
   - Only rows with True values (missing data) are retained. For each position in the mask, pandas examine the Boolean value and make a keep/discard decision: If True,

keep this row (data is problematic); If False, **discard** this row (data is valid).

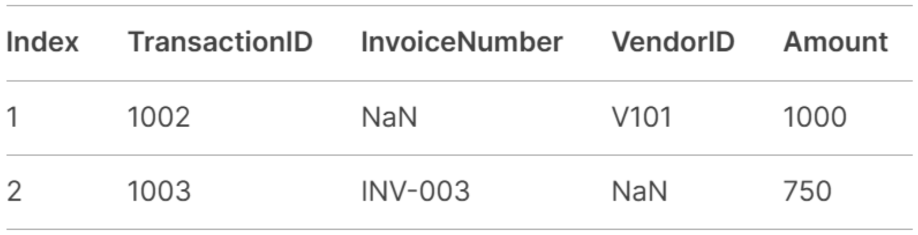


Index	TransactionID	InvoiceNumber	VendorID	Amount
0	1001	INV-001	V100	500
1	1002	NaN	V101	1000
2	1003	INV-003	NaN	750

Boolean mask becomes: **[*False*, True, True]**

- Creates a new DataFrame containing just the problematic records

Resulting missing_data:



Index	TransactionID	InvoiceNumber	VendorID	Amount
1	1002	NaN	V101	1000
2	1003	INV-003	NaN	750