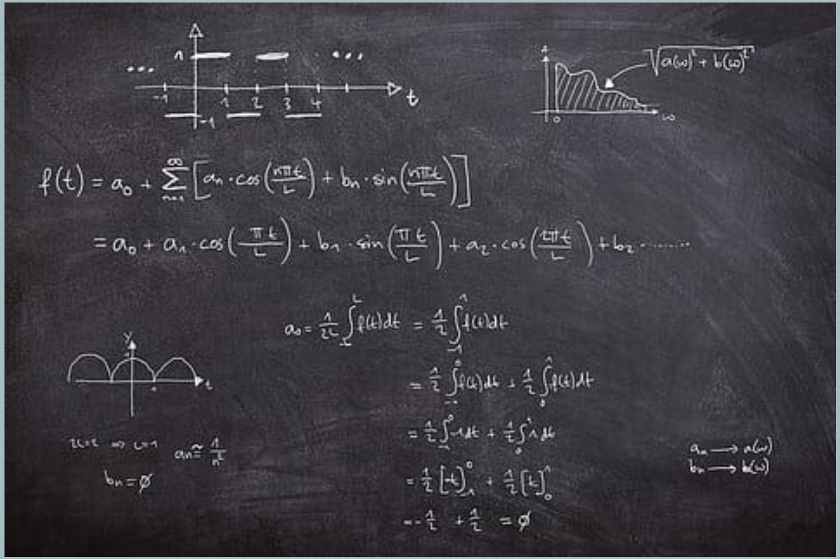
Python's Role in Actuarial Science



By Chris Ixtabalan

Python’s Background

- Guido Van Rossum created Python in 1991.
- Many languages at the time were complicated which meant they were less flexible and harder to read.
- Python was designed to be easy to read and write.

Why I picked Python

- Relevant to my actuarial career
- <u>What actuaries do:</u>
  - Analyze complex data sets
  - Build predictive models
  - Perform quantitative risk assessments
  - Usually by hand
- <u>What Python offers:</u>
  - Data analysis for mortality rates, insurance claims, financial projections
  - Many libraries for statistical modeling
  - Produce accurate data and avoid human error
  - Automate repetitive tasks

Python's Features

- <u>Simplicity and readability:</u>
  - Indentions used as code blocks
    - Prevents errors and data leaks
  - Variables type doesn't need to be declared
  - Interpreted
    - Debugging is made easier
  - "Batteries included"
    - Already comes with libraries which have modules and functions
  - Error handling
- <u>Supports many ways of programming</u>
  - Procedural programming
  - Object-oriented programming
  - Polymorphism
  - Functional Programming

```python
print("Hello, world!")
```

```cpp
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello world!" << endl;
    return 0;
}
```

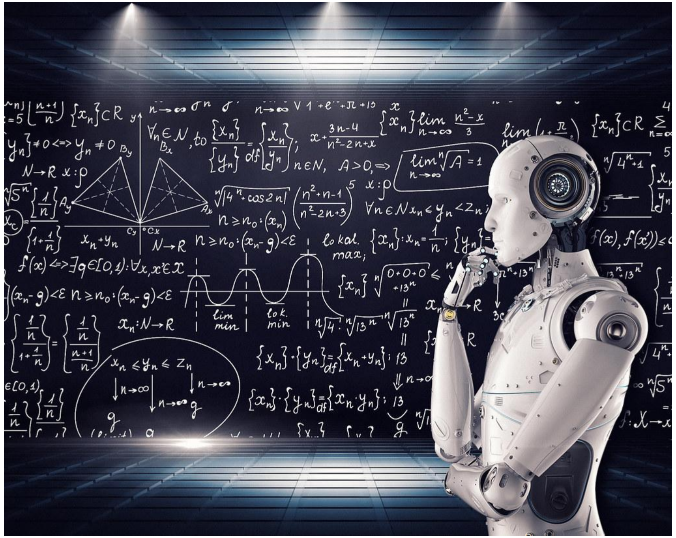
```asm
; Example of 64-bit PE program

format PE64 GUI
entry start

section '.text' code readable executable

  start:
        sub     rsp,8*5         ; reserve stack for API use and make stack dqword aligned

        mov     r9d,0
        lea     r8,[_caption]
        lea     rdx,[_message]
        mov     rcx,0
        call    [MessageBoxA]

        mov     ecx,eax
        call    [ExitProcess]

section '.data' data readable writeable

  _caption db 'Win64 assembly program',0
  _message db 'Hello World!',0

section '.idata' import data readable writeable

  dd 0,0,0,RVA kernel_name,RVA kernel_table
  dd 0,0,0,RVA user_name,RVA user_table
  dd 0,0,0,0,0

  kernel_table:
    ExitProcess dq RVA _ExitProcess
    dq 0
  user_table:
    MessageBoxA dq RVA _MessageBoxA
    dq 0

  kernel_name db 'KERNEL32.DLL',0
  user_name db 'USER32.DLL',0

  _ExitProcess dw 0
    db 'ExitProcess',0
  _MessageBoxA dw 0
    db 'MessageBoxA',0
```

Python’s Scorecard

Characteristic	**Readability**	**Writability**	**Reliability**
Simplicity	High	High	Moderate
Orthogonality	Moderate	Moderate	Moderate
Data Types	High	High	High
Syntax Design	High	High	High
Support for Abstraction	High	High	High
Expressivity	High	High	Moderate
Type Checking	Dynamic	Dynamic	Moderate
Exception Handling	Robust	Robust	High

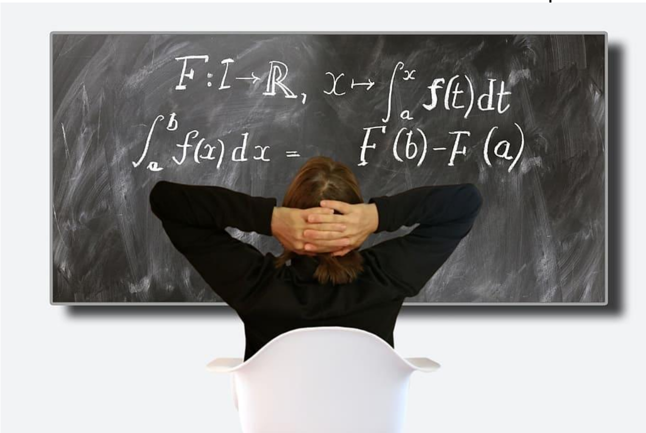
What Python is Typically Used For

- Data analysis
  - Gives efficient tools for data mining and numerical calculations
- Data visualization
  - Libraries can help with data cleaning for large datasets which prepares data visualization
- Machine learning and Artificial Intelligence
  - Help complex neural networking for deep learning



Why I Love Python

- <u>How Python could apply to actuarial science:</u>
  - OOP Could help with modeling insurance policies, financial instruments, and customer
  - Used to predict probabilities, customer behavior, and market trends
  - Functional programming simplifies data by processing collective data, so formulas could easily be applied
  - Complex mathematical calculations are made simple to solve
- <u>Why other actuaries should adapt to the technology</u>
  - Computational calculation is becoming trending
  - Easy to learn
  - They can work as teams
  - Actuarial models can be reused



How I Think it Should be Applied

Insurance Scenario:

An insurance company offers a special 5-year term life insurance policy. The policy pays a benefit based on the year of death within the term:

- **Year 1**: Pays **$1,000**
- **Year 2**: Pays **$2,000**
- **Year 3**: Pays **$3,000**
- **Year 4**: Pays **$4,000**
- **Year 5**: Pays **$5,000**

The probability that the insured dies in any given year is **5%** (0.05), independent of other years.

Question: What is the **expected benefit** paid under this policy?

Coding Plan

- Make a Geometric Formula models for mathematical calculation
- Make a simulation of 1000 policyholders using threading, conditional statements, and classes
- Create exception handling in case of errors
- Use the “random” and “threading” libraries

Geometric Probability Distribution

$$P(X = n) \;=\; p(1-p)^{n-1}$$

$$P(X > n) \;=\; (1-p)^{n}$$

Mean $\mu = \frac{1}{p}$

Variance $\sigma^2 = \frac{1-p}{p^2}$

p – probability of success

n- number of first successful trial

Demonstration

Importing Python Libraries

```
import random
#Imported to enabl
import threading
```

Using Classes to represent a policyholder

```
class PolicyHolder:
    #Used to represent a policyholder
    def __init__(self, policy_id):
        self.policy_id = policy_id #Their ID
        self.benefit = 0 #Their starting benefit

    #Simulates life of policyholder
    def simulate_life(self):
        probability_of_death = 0.05  # 5% chance of dea
        for year in range(1, 6):  # Years 1 to 5
            #Generates random number 0-1, if number is
            if random.random() < probability_of_death:
                self.benefit = 1000 * year  # Benefit i
                break  # Policyholder dies; exit the lo
```

Simulation Function

```
def simulate_policies_threaded(num_simulations, num_threads=4):
    total_benefit = 0 #Total sum of all benefits for all policyholders
    benefits = [] #List to store individual beneftis of each policyholder
    lock = threading.Lock()  # Lock: To make sure threadts don't interfere with each c

    #Worker function that will be runned in each thread
    def worker(simulations_per_thread, thread_id):
        nonlocal total_benefit, benefits #Allows variable modifcation from outer funct
        thread_benefit = 0 #Total benefit for each specific thread
        thread_benefits = [] #Lists of each benefit

        #Ran for each simulation
        for i in range(simulations_per_thread):
            policyholder = PolicyHolder(f"Policy_{thread_id}_{i}") #Creates ID
            policyholder.simulate_life() #Simulate life of policyholder
            thread_benefit += policyholder.benefit #Add policyholder's benefit to thre
            thread_benefits.append(policyholder.benefit) #Then stores it into the thre
        with lock:
            total_benefit += thread_benefit #Adds a threads benefit to overall total b
            benefits.extend(thread_benefits) #Same thing but to overall list

    simulations_per_thread = num_simulations // num_threads #Divides total simulation
    threads = [] #List to store all threads

    #Creates and start threads
    for thread_id in range(num_threads):
        t = threading.Thread(target=worker, args=(simulations_per_thread, thread_id))
        threads.append(t) #Add thread to list of threads
        t.start() #Starts thread

    # Wait for all threads to complete
    for t in threads:
        t.join() #Waits until thread "t" finishes

    #Calculates average beneftt
    expected_benefit = total_benefit / num_simulations
    #Returns all individuals outputs
    return expected_benefit, benefits
```

Demonstration

Geometric Probability Function

```
def calculate_analytical_expected_benefit():
    probability_of_death = 0.05
    expected_benefit = 0
    for year in range(1, 6):  # Years 1 to 5
        benefit = 1000 * year
        # Calculate the probability of dying in a specific year
        prob = (1 - probability_of_death) ** (year - 1) * probability_of_death
        expected_benefit += benefit * prob  # Sum up the expected benefits
    return expected_benefit
```

Geometric Probability Distribution

$$P(X = n) \;=\; p(1-p)^{n-1}$$

$$P(X > n) \;=\; (1-p)^{n}$$

Mean $\mu = \frac{1}{p}$

Variance $\sigma^2 = \frac{1-p}{p^2}$

p – probability of success
n- number of first successful trial

Main Function w/ other formulas

```
def main():
    try:
        num_simulations = 100000  # Number of simulations
        #Calculate using simulation with threads
        expected_benefit_simulation, benefits = simulate_policies_threaded(num_simulations)
        #Calculate using Geometric Distribution
        expected_benefit_analytical = calculate_analytical_expected_benefit()

        ########################Results###################
        print(f"Expected Benefit Using Math Concepts: ${expected_benefit_analytical:.2f}")
        print(f"Expected Benefit Using Simulations: ${expected_benefit_simulation:.2f}")

        # Additional analysis using the benefits list
        # Calculate variance and standard deviation
        mean = expected_benefit_simulation
        variance = sum((x - mean) ** 2 for x in benefits) / num_simulations
        std_dev = variance ** 0.5
        print(f"Simulated Variance: ${variance:.2f}")
        print(f"Simulated Standard Deviation: ${std_dev:.2f}")
        ################################################
    #Exception Handling
    except Exception as e:
        print("An error occurred during the simulation.")
        print("Error message:", str(e))
```

Conclusion

Final Outcome

```
Expected Benefit Using Math Concepts: $655.48
Expected Benefit Using Simulations: $650.35
Simulated Variance: $1906334.88
Simulated Standard Deviation: $1380.70
christopherixtabalan@Chris-OIs-Mac Codes %
```

- Python is a versatile and flexible programming language for professionals
- It has changed the way people do problem solving
- Making actuarial models can help create accurate data and find probabilities
- Great for mathematical computations and data analysis