01a_python_basics

May 9, 2025

1 Python for Actuaries Part 1

1.1 Agenda

In this notebook, we will cover: - Introduction to Python - IPYNB: Interactive Python Notebooks - Data types and variables - Loops and control structures

1.2 Introduction to Python and Notebooks

Interactive Python notebooks consist of cells. Each cell contains either code or text. A text cell (like this one) contains text and can be formatted using **markdown**. A double-click on the cell makes it editable.

A code cell (like the next one) contains code. Typically Python code (but much more is possible: R, Java, Julia, SQL, …). To execute a code cell, you can press the "Play" button to the left of the cell or use the keyboard shortcut Ctrl + Enter within the cell.

Hello World in Python:

[ ]:
```
print("Hello World")
```

When executing a cell in a notebook, the value that results from evaluating the *last expression* in the cell is displayed as the "result" (by default). If the last construct in a cell is not an expression (e.g., a statement), or if the expression evaluates to `None` (more on this later), then no result is displayed.

[ ]:
```
3 + 3
2 + 5
```

The *output* (e.g., generated by `print`) is displayed separately from the result of the cell. Multiple outputs are collected and displayed in the same output area for a specific cell.

[ ]:
```
print(3+3)
print(2+5)
2 * 3
```

If we want to see the result of multiple expressions in a cell, we often group the expressions in parentheses, resulting in an aggregated value known as a *tuple*.

[ ]:
```
(1+2, 2*3, 4/5)
```

[ ]:
```
(6*7, not True, 'hello' + 'world')
```

2 Numbers

[ ]:
```
# int: integers, unlimited precision
(
    1,
    500,
    -123456789,
    659829378498273987498273 4
)
```

[ ]:
```
# basic operations
(
    1 + 2,
    1 - 2,
    2 * 3,
    2 * 3 + 2 * 4,
    2 / 5,
    2 ** 3, # exponentiation
    abs(-25)
)
```

3 Truth Values

[ ]:
```
(
    True,
    False
)
```

[ ]:
```
(
    True and True,
    False and True,
    True and False,
    False and False
)
```

[ ]:
```
(
    True or True,
    False or True,
    True or False,
    False or False
)
```

[ ]:
```
# relational operators
(
```

```
    1 == 1,
    1 != 2,
    1 < 2,
    1 <= 1,
    1 > 0,
    1 >= 1,
    1.0 == 1,
    1.00000000001 == 1,
    type(1) == type(1.0)
)
```

[ ]:
```
# chained relational operators
x = 10
y = 20
z = 30
(
    0 <= x < 100,
    0 <= x and x < 100,
    x < y < z < 40,
    x <  y and y < z and z < 40,
    x < z > y,
    x < z and z > y
)
```

[ ]:
```
# object identity (reference) testing
x = 1000
y = 1000
(
    x == x,   # value comparison
    x is x,   # identity comparison
    x == y,
    x is y,
    id(x) == id(y) # `id` returns the memory address (aka "identity") of an␣
 ↪object
)
id(x)
```

[ ]:
```
print(id(y), id(x))
```

[ ]:
```
# but Python caches small integers! so ...
x = 5
y = 5
print(x is y)
y = 6
print(x is y)
```

4 Basic Types

In Python, variables do not have *types*! Values have types, but these are not explicitly declared. A variable can take on different type representations during its lifetime:

[ ]:
```
a = 2 # starts out an integer
print(type(a)) # the `type` function tells us the type of a value

a = 1.5
print(type(a))

a = 'hello'
print(type(a))
```

4.0.1 Data Types

- Text Type: str
- Numeric Types: int, float, complex
- Sequence Types: list, tuple, range
- Mapping Type: dict
- Set Types: set, frozenset
- Boolean Type: bool
- Binary Types: bytes, bytearray, memoryview

See also https://docs.python.org/3/library/stdtypes.html

[ ]:
```
x = 5
print(type(x))
x = "Hello World"
print(type(x))
x = 20.5
print(type(x))
x = 1j
print(type(x))
x = ["apple", "banana", "cherry"]
print(type(x))
x = ("apple", "banana", "cherry")
print(type(x))
x = range(6)
print(type(x))
x = {"name" : "John", "age" : 36}
print(type(x))
x = {"apple", "banana", "cherry"}
print(type(x))
x = True
print(type(x))
```

4.1 Simplifications

[ ]:
```
a = 0
a += 2
a *=3
a
```

[ ]:
```
# easy python "swap"

a, b = 'apples', 'bananas'
a, b = b, a
```

5 If-Else

Attention: In Python, there are no block start-end markers, such as brackets or similar. It is done with indentations (4 spaces or 1 tab).

[ ]:
```
x = 2
if x == 2:
    print("x is equal to 2")
else:
    print("x not equal to 2")
```

[ ]:
```
score = 70
grade = None

if score >= 90:
    grade = 'A'
elif score >= 80:
    grade = 'B'
elif score >= 70:
    grade = 'C'
elif score >= 60:
    grade = 'D'
else:
    grade = 'E'

(score, grade)
```

6 Loops

In Python, you can iterate over just about anything (as long as it is iterable). We can iterate over lists, tuples, strings, or even ranges. A simple for loop looks like this:

[ ]:
```
for i in range(5): # range(n) creates a list of numbers from 0 to n-1
    print(i) # watch out for the indentation!
```

[ ]:
```
for c in "hello world":
    print(c)
```

[ ]:
```
for x in (a,1,True):
    print(x)
```

While Loop

[ ]:
```
i = 10
s = 0
while i!=0:
  s = s + i
  i = i - 1
print(s)
```

6.1 break, continue, pass in Python

These statements help control the flow of loops.

6.1.1 break

The break statement terminates the loop prematurely as soon as a condition is met.

[ ]:
```
print("Example for 'break':")
for i in range(1, 11):
    if i == 5:
        print("iteration is 5, break")
        break  # loop will break when i is 5
    print(i)
```

6.1.2 continue

The continue statement skips the rest of the current loop iteration and continues with the next iteration.

[ ]:
```
print("\Example for'continue':")
for i in range(1, 11):
    if i % 2 == 0:
        continue  # skip even numbers
    print(i)  #  print odd numbers only
```

6.2 pass

The pass statement does nothing; it is often used as a placeholder in loops or conditions when a block is syntactically required but no logic has been inserted yet.

Here in the course, `pass` often means something else as well: It's your turn. In some places, pass indicates that you should replace these lines of code with appropriate code.

[ ]:
```
print("example for 'pass':")
for i in range(1, 11):
    if i % 2 == 0:
        pass  # do nothing for even numbers
    else:
        print(i)  # print odd numbers only
```

7 Tasks

Please complete the following tasks.

7.1 Task 1 (Multiplication Table)

Write a program that generates and displays a multiplication table for the numbers 1 to 10. The program should use loops and conditional statements to calculate and output the table.

Note: with `f"{number_variable:4}"`, you can control that the values of the variable `number_variable` are displayed with 4 spaces.

Desired output:

```
Multiplication table for 1 to 10:
   1    2    3    4    5    6    7    8    9   10
   2    4    6    8   10   12   14   16   18   20
   3    6    9   12   15   18   21   24   27   30
   4    8   12   16   20   24   28   32   36   40
   5   10   15   20   25   30   35   40   45   50
   6   12   18   24   30   36   42   48   54   60
   7   14   21   28   35   42   49   56   63   70
   8   16   24   32   40   48   56   64   72   80
   9   18   27   36   45   54   63   72   81   90
  10   20   30   40   50   60   70   80   90  100
```

[4]:
```
#  Initialize
n = 10
# Print the multiplication table for numbers from 1 to n
print(f"Table for 1 up to {n}:")

for i in range(1, n + 1):
    for j in range(1, 11):
        print(f"{i * j:4}", end="")
    print()  # Print a blank line between tables
```

```
Table for 1 up to 10:
   1   2   3   4   5   6   7   8   9  10
   2   4   6   8  10  12  14  16  18  20
   3   6   9  12  15  18  21  24  27  30
   4   8  12  16  20  24  28  32  36  40
   5  10  15  20  25  30  35  40  45  50
```

```
 6  12  18  24  30  36  42  48  54  60
 7  14  21  28  35  42  49  56  63  70
 8  16  24  32  40  48  56  64  72  80
 9  18  27  36  45  54  63  72  81  90
10  20  30  40  50  60  70  80  90 100
```

8 Task 2 (Counting Game)

Write a program that counts from 1 to a defined number n and checks certain conditions. If the number is divisible by 3, the program should output "Fizz"; if the number is divisible by 5, it should output "Buzz"; and if the number is divisible by both 3 and 5, it should output "FizzBuzz".

Note: You can use `%` for modulo calculations. For example, `(6 % 3 == 0)` evaluates to `True`.

Example output:

```
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
...
```

[10]:
```python
# Initialization
n = 15

# counting and checking
for i in range(1, n + 1):
    if i % 3 == 0 and i % 5 == 0:
        print("FizzBuzz")
    elif i % 3 == 0:
        print("Fizz")
    elif i % 5 == 0:
        print("Buzz")
    else:
        print(i)
```

```
1
2
Fizz
4
Buzz
Fizz
7
```

```
8
Fizz
Buzz
11
Fizz
13
14
FizzBuzz
```

8.1 Task 3 (Finding and Displaying Prime Numbers)

Write a program that finds and displays all prime numbers between 1 and a defined number n.

Example output for the defined number 15:

```
Prime numbers between 1 and 15:
2
3
5
7
11
13
```

[11]:
```python
# initialization
n = 50

# Check for primes and print them
print(f"Primes between 1 and {n}:")

for num in range(2, n + 1):
    is_prime = True
    for i in range(2, int(num ** 0.5) + 1):
        if num % i == 0:
            is_prime = False
            break
    if is_prime:
        print(num)
```

```
Primes between 1 and 50:
2
3
5
7
11
13
17
19
23
29
```

```
31
37
41
43
47
```