Functions

What and Why?

- In Python, a function is some reusable code that takes arguments(s) as input, does some computation, and then returns a result or results
- We define a function using the def reserved word
- We call/invoke the function by using the function name, parentheses, and arguments in an expression

Type Conversions

- When you put an integer and floating point in an expression, the integer is implicitly converted to a float. This is called type promotion.
- You can control this with the built-in functions int() and float()

```
>>> print(float(99) + 100)
199.0
>>> i = 42
>>> type(i)
<class'int'>
>>> f = float(i)
>>> print(f)
42.0
>>> type(f)
<class'float'>
>>>
```

String Conversions

- You can also use int() and float() to convert between strings and integers
- You will get an **error** if the string does not contain numeric characters

```
>>> sval = '123'
>>> type(sval)
<class 'str'>
>>> print(sval + 1)
Traceback (most recent call last):  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
>>> ival = int(sval)
>>> type(ival)
<class 'int'>
>>> print(ival + 1)
124
>>> nsv = 'hello bob'
>>> niv = int(nsv)
Traceback (most recent call last):  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'x'
```

Python Functions

- There are two kinds of functions in Python.
  - Built-in functions that are provided as part of Python - `print()`, `input()`, `type()`, `float()`, `int()` …
  - Functions that we define ourselves and then use
- We treat function names as "new" reserved words (i.e., we avoid them as variable names)

sin(x), cos(x), log(x) functions are reserved for the specific operations, no other function possible.

In Python, you can define your own functions at ease and give it a meaningful name and use them **ordinarily**.

Collection of Statements Under a Logical Name

```
def fun():
    print("IIT Gandhinagar")
    print("IIT Hyderabad")

fun()
```

The **def** says that whatever we are writing next is a block of code. This block of code will be collectively called under a name **fun**(). The parentheses '()' are important to distinguish it from a regular variable (non-function)

Indentation (spaces, tabs) and Colon ':' MUST be respected syntactically

Body of the function. These are the statements that will be executed when fun() is called/invoked later somewhere

The definition/declaration of a function MUST be written before it is called/invoked/used

Otherwise

```
Traceback (most recent call last):
  File "main.py", line 1, in <module>
    fun()
NameError: name 'fun' is not defined
```

Functions that Consume, Produce, and Return

```
def f(k):
  res = k * k
  return res

x = int(input())
y = f(x)
print(y)
```

RETURN

CALL

argument 'x' is passed to f().

The control reaches to the parameter 'k'. Executes k = x at this point (intuitively).

The return statement (from callee) transfers back control to the call site (to caller)

Starting point of the entire program is the input() function

The return clause is optional, as well as any of its **arguments**

What will happen if we omit the return statement altogether?

A void function does not return

Functions that Consume, Produce, and Return

```
def f(k):
  res = k * k
  return res

x = int(input())
y = f(x)
print(y)
```

RETURN

CALL

```
def fun(name):
    print(f"Hello, {name}!")

fun("IIT Gandhinagar")
```

Output: Hello, IIT Gandhinagar!

A void function does not return

Common Math Functions

Basic: math.sqrt(x), math.pow(x, y), math.exp(x) etc.

Trigonometric: math.sin(x), math.radians(x) etc.

Logarithmic: math.log10(x), math.log(x[, base]) etc.

Constants: math.pi, math.e etc.

Others: math.factorial(x), math.ceil(x), math.floor(x) etc.

```
>>>print(math)
<module ‘math’(built-in)>
```

```
import math

print("Square root of 16:", math.sqrt(16))
print("2 raised to the power of 3:", math.pow(2, 3))
```

```
Output:
4.0
8.0
```

Math Functions and Random Numbers

```
import math

def calculate_snr(signal_power, noise_power):
    if noise_power == 0:
        raise ValueError("Noise power cannot be zero.")

    snr = signal_power / noise_power
    snr_db = 10 * math.log10(snr)
    return snr, snr_db

signal_power = 100
noise_power = 10

snr, snr_db = calculate_snr(signal_power, noise_power)
print(f"SNR: {snr} (linear scale)")
print(f"SNR: {snr_db} dB")
```

```
class CustomError(Exception):
    pass

raise CustomError("This is a custom
error message.")
```

```
try:
    # Code that may raise an
exception
    signal_power = 100
    noise_power = 0  # This will
trigger an error
    snr = signal_power / noise_power
except ValueError as e:
    print(f"Error: {e}")
```

```
import random
for i in range(10):
    x = random.random()
    print(x)
```

Functions with multiple parameters/arguments

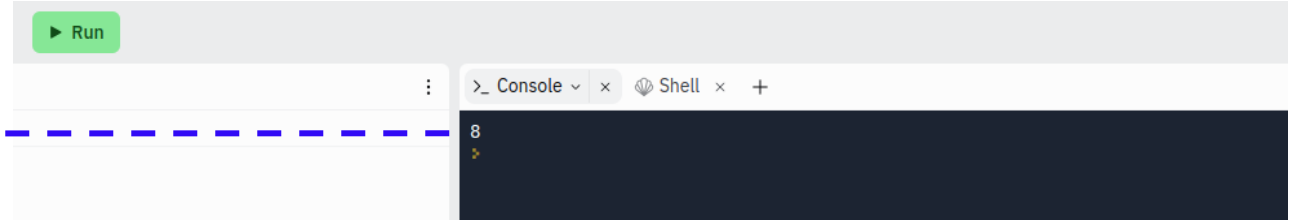
- We can define more than one parameter in the function definition
- We simply add more arguments when we call the function
- We match the number and order of arguments and parameters

```
def addtwo(a, b):
  sum = a + b
  return sum

x = addtwo(3, 5)
print(x)
```

```
8
```

Bijective mapping of argument and parameters is strictly in-order from left to right



To Function or Not To Function…

- Organize your code into “paragraphs” - capture a complete thought and “name it”
- Don’t repeat yourself - make it work once and then reuse it
- If something gets too long or complex, break it up into logical chunks and put those chunks in functions
- Make a library of common stuff that you do over and over - perhaps share this with your friends…

Why function: take away

- Make program easier to read, understand, and debug.
- Make program a smaller and effective by eliminating the repetitive code and changes.
- Dividing a long program into function allows you to debug the parts one at a time and then assemble them into a working whole.
- Well-designed function are often useful for many programs. Once you write and debug one, you can reuse it.

Iteration and Loops

Conditional Steps: Recap

Program

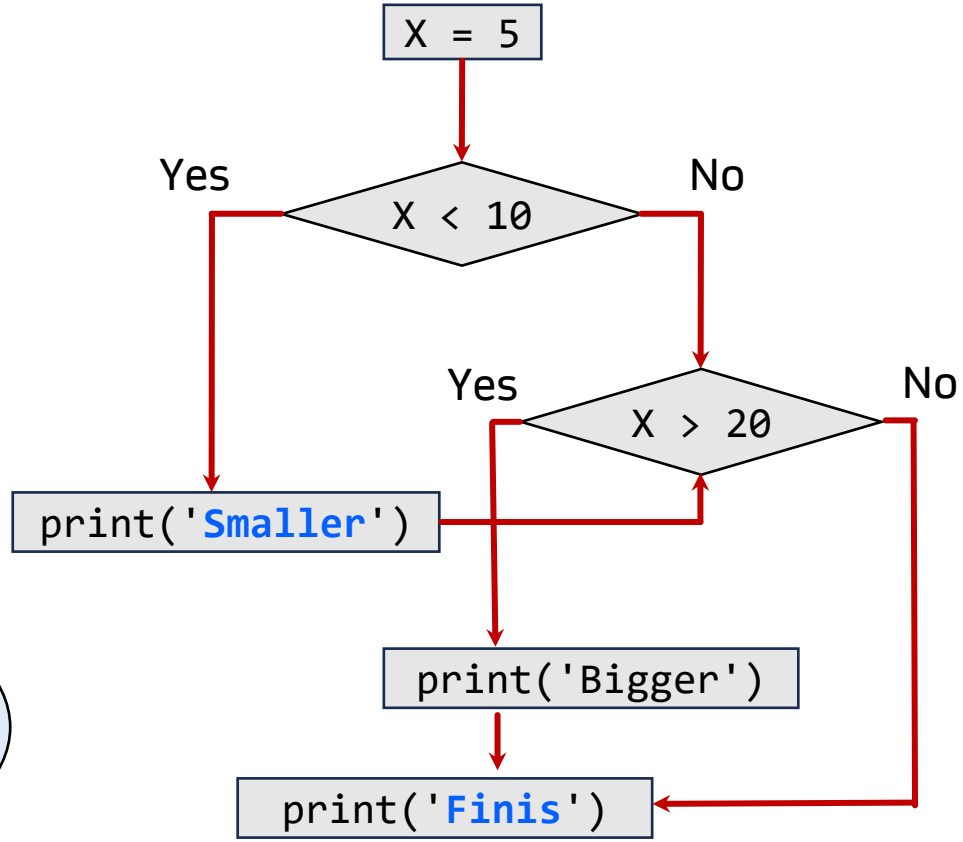
```
x = 5
if x < 10:
    print('Smaller')
if x > 20:
    print('Bigger')

print('Finis')
```

Indentation (spaces, tabs) MUST be consistent.

Colon ':' is a MUST part of the syntax.

Control flow



Output

```
Smaller
Finis
```

When a program is running, **the execution path is not unique**. The non-uniqueness comes from **decision making** on which path to take!

Repeated Steps: Infinite Loop

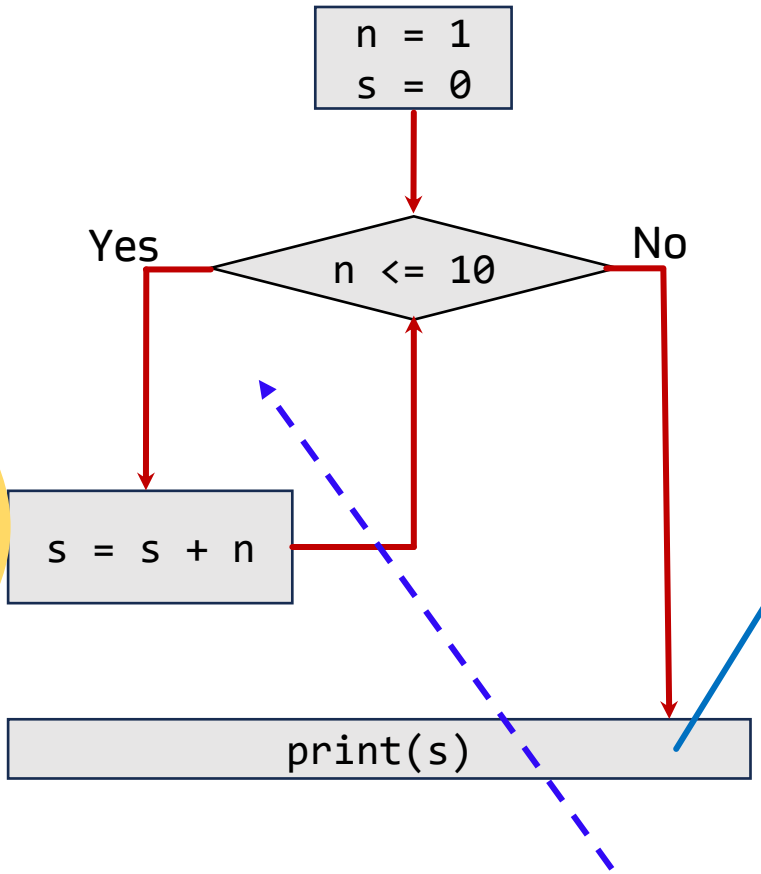
Program

```
n = 1
s = 0
while n <= 10:
    s = s + n
print(s)
```

What will be the output of the program?

Indentation (spaces, tabs) and Colon ':' MUST be respected syntactically

Control flow



This loop rotates forever (infinite loop).

Output

?

When a program is running, **the execution of some instruction can be repeated**. This called looping. Here the **iteration variable** is **n**

Repeated Steps: Finite Loop

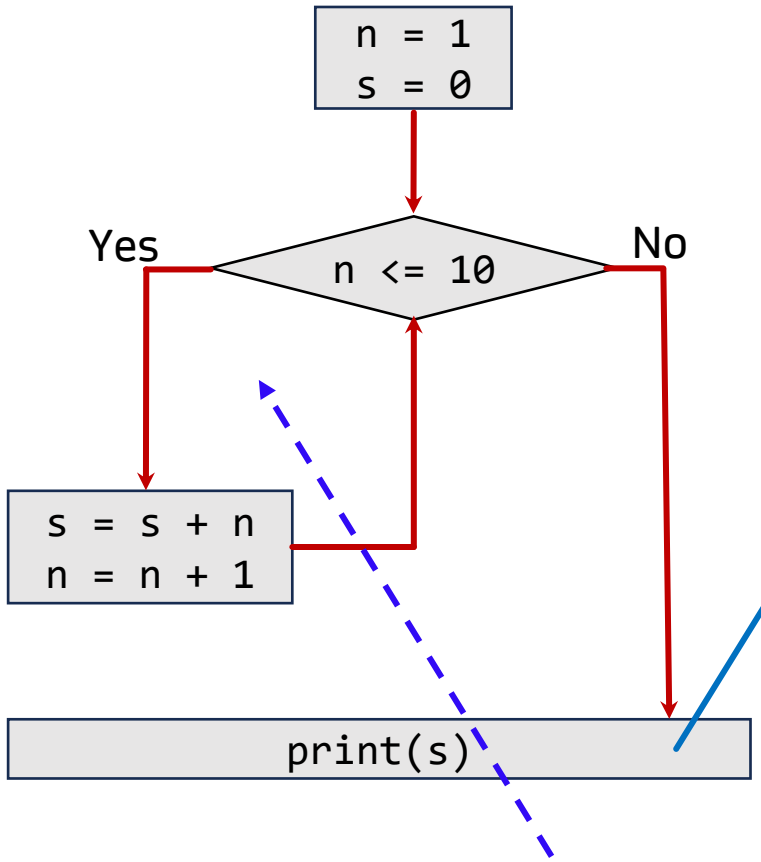
Program

```
n = 1
s = 0
while n <= 10:
    s = s + n
    n = n + 1
print(s)
```

How many times will the loop condition be evaluated?

n+1;
n times for true, once for false

Control flow



Output

```
55
```

When a program is running, **the execution of some instruction can be repeated**. This called looping. Here the **iteration variable** is **n**

Now this loop becomes finite, and the body will execute n times

Breaking Out of a Loop

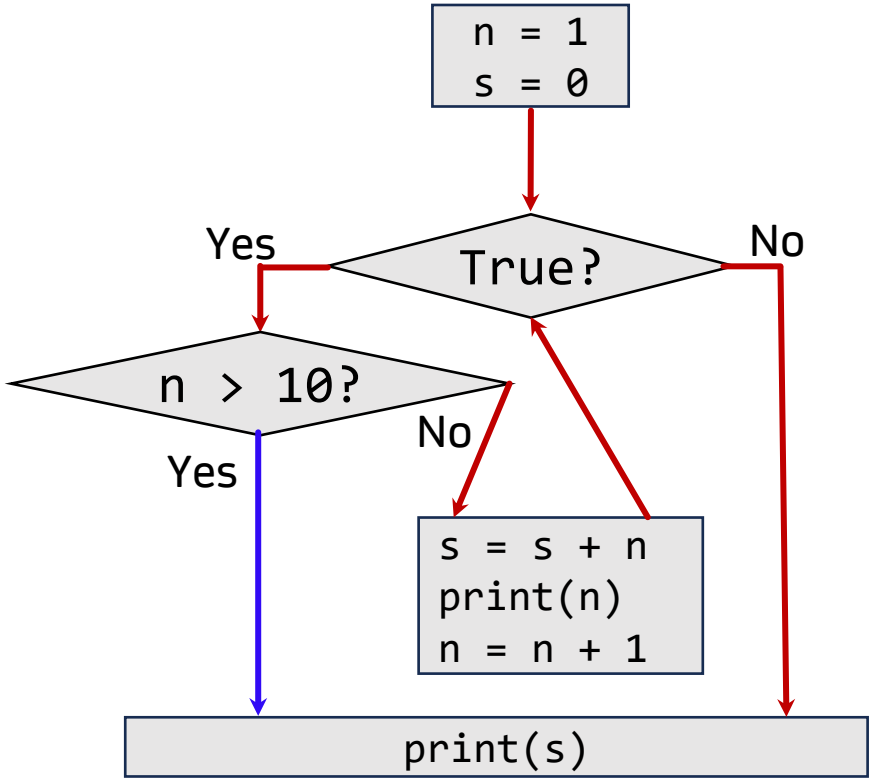
- The break statement ends the current loop and jumps to the statement immediately following the loop
- It is like a loop test that can happen anywhere in the body of the loop

Breaking Out of a Loop: Control Flow

Program

```
n = 1
s = 0
while True:
  if n > 10:
    break
  s = s + n
  print(n)
  n = n + 1
print(s)
```

Control flow



Output

```
1
2
3
4
5
6
7
8
9
10
55
```

Breaking Out of a Loop

Program

```
n = 1
s = 0
while True:
  if n > 10:
    break
  s = s + n
  print(n)
  n = n + 1
print(s)
```

Output

```
1
2
3
4
5
6
7
8
9
10
55
```

What will happen if we change > to >=, <, <=?

Program

```
n = 1
s = 0
while True:
  if n >= 10:
    break
  s = s + n
  print(n)
  n = n + 1
print(s)
```

Output

```
1
2
3
4
5
6
7
8
9
45
```

Breaking Out of a Loop

Program

```
n = 1
s = 0
while True:
  if n < 10:
    break
  s = s + n
  print(n)
  n = n + 1
print(s)
```

Output

```
0
```

Program

```
n = 1
s = 0
while True:
  if n <= 10:
    break
  s = s + n
  print(n)
  n = n + 1
print(s)
```

Output

```
0
```

String Conversions: Recall

- You can also use int() and float() to convert between strings and integers
- You will get an **error** if the string does not contain numeric characters

```
>>> sval = '123'
>>> type(sval)
<class 'str'>
>>> print(sval + 1)
Traceback (most recent call last):  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
>>> ival = int(sval)
>>> type(ival)
<class 'int'>
>>> print(ival + 1)
124
>>> nsv = 'hello bob'
>>> niv = int(nsv)
Traceback (most recent call last):  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'x'
```

How do string conversions affect the termination of indefinite loops accepting user input?

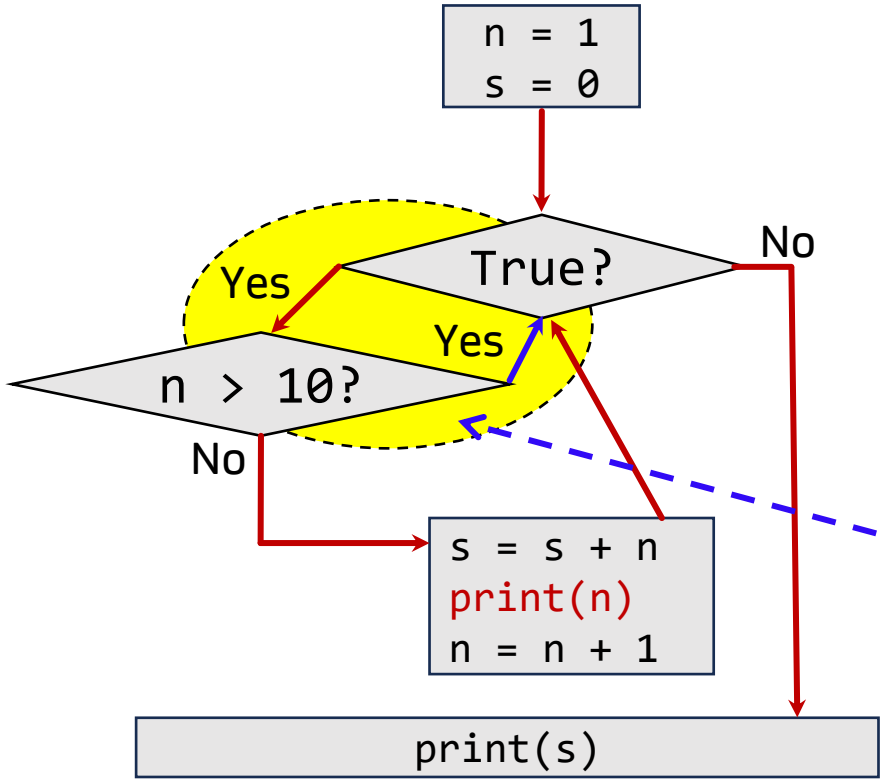
Finishing an Iteration with continue

- The continue statement ends the current iteration and jumps to the top of the loop and starts the next iteration.

Program

```
n = 1
s = 0
while True:
  if n > 10:
    continue
  s = s + n
  print(n)
  n = n + 1
print(s)
```

Control flow



Output

```
1
2
3
4
5
6
7
8
9
10
--
```

Finishing an Iteration with continue

Program

```
n = 1
s = 0
while True:
  if n > 10:
    continue
  s = s + n
  print(n)
  n = n + 1
print(s)
```

Output

```
1
2
3
4
5
6
7
8
9
10
--
```

What will happen if we change > to >=, <, <=?

Program

```
n = 1
s = 0
while True:
  if n >= 10:
    continue
  s = s + n
  print(n)
  n = n + 1
print(s)
```

Output

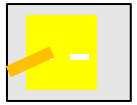
```
1
2
3
4
5
6
7
8
9
--
```

Finishing an Iteration with continue

Program

```
n = 1
s = 0
while True:
  if n < 10:
    continue
  s = s + n
  print(n)
  n = n + 1
print(s)
```

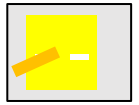
Output



Program

```
n = 1
s = 0
while True:
  if n <= 10:
    continue
  s = s + n
  print(n)
  n = n + 1
print(s)
```

Output



Indefinite Loops

- While loops are called "indefinite loops" because they keep going until a logical condition becomes False
- The loops we have seen so far are pretty easy to examine to see if they will terminate or if they will be "infinite loops"
- Sometimes it is a little harder to be sure if a loop will terminate (until program is executed)

```
while True:
  n = int(input())
  if n == 0:
    break
```

What will be the output of the program, if you enter ints/floats?

Acknowledgements / Contributions

Initial Development: Charles Severance, University of Michigan School of Information

Contributors 2024 - Yogesh K. Meena and Shouvick Mondal, IIT Gandhinagar