Data 301 Data Analytics
Python II

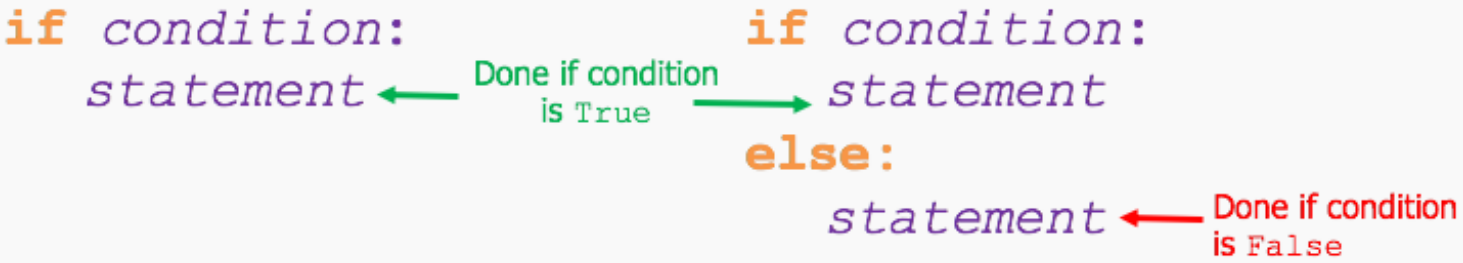
Dr. Irene Vrbik

University of British Columbia Okanagan
irene.vrbik@ubc.ca

Term 1, 2019

Decisions

Decisions allow the program to perform different actions based on conditions. Python decision syntax:



- The statement after the if condition is only performed if the condition is True.
- If there is an else, the statement after the else is done if condition is False.
- Indentation is important! Remember the colon!

Decisions if/elif Syntax

If there are more than two choices, use the `if/elif/else` syntax:

```
if condition:
    statement
elif condition:
    statement
elif condition:
    statement
else:
    statement
```

```
if n == 1:
    print("one")
elif n == 2:
    print("two")
elif n == 3:
    print("three")
else:
    print("Too big!")
print("Done!")
```

Note that there can be multiple `elif` statements but there can only be up to one `else` statement.

Decisions if/elif Syntax

Once a condition is met, no subsequent conditions are checked:

```
1  n = 1
2  if n == 1:
3      print("one")
4  elif n>0: # this condition is never checked since
5      # the condition on line 2 has already been satisfied
6      print("positive number")
7  elif n == 3:
8      print("three")
9  else:
10     print("Too big!")
11 print("and Done!") #  not part of the if statement
```

The above returns:

```
one
and Done!
```

Decisions if/elif Syntax

Once a condition is met, no subsequent conditions are checked:

```
1  n = 3
2  if n == 1:
3      print("one")
4  elif n>0:
5      print("positive number")
6  elif n == 3: # this condition is never checked since
7      # the condition on line 4 has already been satisfied
8      print("three")
9  else:
10     print("Too big!")
11 print("and Done!") #  not part of the if statement
```

The above returns:

```
positive number
and Done!
```

Decision Block Syntax

Statements executed after a decision in an `if` statement are indented for readability. This indentation is also how Python knows which statements are part of the block of statements to be executed.

- If you have more than one statement, make sure to indent them. Be consistent with either using tabs or spaces. Do not mix them!

- ```
  if age > 19 and name > "N":
      print("Not a teenager")
      print("Name larger than N")
  else:
      print("This is statement #1")
      print(" and here is statement #2!")
  ```
  ```

Decisions if/elif Syntax

Check out the difference for age = 20:

```
age = 20
if age > 19:
    print("Not a teenager")
    print("Sorry")
else:
    print("You're young")
    print("ID checked")
```

The above returns:

```
Not a teenager
Sorry
```

```
age = 20
if age > 19:
    print("Not a teenager")
    print("Sorry")
else:
    print("You're young")
print("ID checked")
```

The above returns:

```
Not a teenager
Sorry
ID checked
```

Example 17

What is the output of the following code?

```
n = 3
if n < 1:
    print("one")
elif n > 2:
    print("two")
elif n == 3:
    print("three")
```

A) nothing

B) one

C) two

D) three

E) error

Answer:

What is the output of the following code?

```
n = 3
if n < 1:
    print("one")
elif n > 2:
    print("two")
elif n == 3:
    print("three")
```

A) nothing
B) one
C) two
D) three
E) error

Example 18

What is the output of the following code?

```
n = 3
if n < 1:
    print("one")
elif n >
    print("two")
else:
    print("three")
```

A) nothing

B) one

C) two

D) three

E) error

Answer:

What is the output of the following code?

```
n = 3
if n < 1:
    print("one")
elif n > 2
    print("two")
else:
    print("three")
```

1. nothing
2. one
3. two
4. three
5. error (missing colon)

Example 19

What is the output of the following code?

```
n = 1
if n < 1:
    print("one")
elif n > 2:
    print("two")
else:
    print("three")
print("four")
```

A) nothing

B) one
four

C) three

D) three
four

E) error

Answer:

What is the output of the following code?

```
n = 1
if n < 1:
    print("one")
elif n > 2:
    print("two")
else:
    print("three")
print("four")
```

A) nothing B) one four C) three D) *three four* E) error

Example 20

What is the output of the following code?

```
n = 0
if n < 1:
    print("one")
    print("five")
elif n == 0:
    print("zero")
else:
    print("three")
print("four")
```

A) nothing

B) one
four

C) one
five
four

D) one
five
zero
four

E) error

Answer:

What is the output of the following code?

```
n = 0
if n < 1:
    print("one")
    print("five")
elif n == 0:
    print("zero")
else:
    print("three")
print("four")
```

A) nothing

B) one
four

C) *one*
five
four

D) one
five
zero
four

E) error

Try it: Decisions

Example 21

Write a Python program that asks the user for a number then prints out if it is even or odd.

Example 22

Write a Python program that asks the user for an integer. If that number is between 1 and 5, prints out the word for that number (e.g. 1 is one). If the number is not in that range, print out error.

Loops and Iteration

A **loop** repeats a set of statements multiple times until some condition is satisfied.

- Each time a loop is executed is called an **iteration**.

A `for` loop repeats statements a certain number of times.

- It will iterate over a sequence, eg. 1, 2, ... 10
- or it could iterate over group/collection elements, eg. lines in a document, elements in a list

A `while` loop repeats statements while a condition is True.

- At each iteration we will check this condition.
- If its `True` we complete another iteration
- If its `False` we exit the loop.

`while` loops

The most basic looping structure is the *while* loop.

A while loop continually executes a set of statements while a condition is true. Syntax:

```
while condition:
    statement1
    statement2
                ⋮
```

Example:

```
n = 1
while n <= 5:
    print(n)
    n = n + 1
```

prints the values 1 through 5.

Shorthand

In addition to the = operator for assigning a value to a variable, Python also supports a shorthand version that compounds various mathematical operators with the assignment operator:

Table: Table taken from this source

Operator	Example	Equivalent to
=	`x = 5`	`x = 5`
+=	`x += 5`	`x = x + 5`
-=	`x -= 5`	`x = x - 5`
*=	`x *= 5`	`x = x * 5`
/=	`x /= 5`	`x = x / 5`
%=	`x %= 5`	`x = x % 5`

Question: `while` loop

Example 23

What is the output of the following code:

```
n = 4
while n >= 0:
    n = n - 1
    print(n)
```

A) numbers 3 to -1

B) numbers 3 to 0

C) numbers 4 to 0

D) numbers 4 to -1

E) numbers 4 to infinity

Question: `while` loop

answer

What is the output of the following code:

```
n = 4
while n >= 0:
    n = n - 1
    print(n)
```

A) *numbers 3 to -1*

B) numbers 3 to 0

C) numbers 4 to 0

D) numbers 4 to -1

E) numbers 4 to infinity

Question: `while` loop 2

Example 24

What is the output of the following code:

```
n = 1
while n <= 5:
    print(n)
n = n + 1
```

A) nothing

B) numbers 1 to 5

C) numbers 1 to 6

D) lots of 1s

Question: `while` loop

answer

What is the output of the following code:

```
n = 1
while n <= 5:
    print(n)
n = n + 1
```

A) nothing

B) numbers 1 to 5

C) numbers 1 to 6

D) *lots of 1s Infinite loop without the fourth line indented*

The `for` loop

- A `for` loop repeats statements a given number of times.
- One way of building a for loop is to iterate over a sequence which we create using `range()`

```
for i in range(1,6):
    print(i)
```

- The above prints the numbers 1–**5**.

`range(start, end)`

In `range(start, end)`, the `start` number in inclusive and the `start` number is exclusive.

Using `range()`

- The general form of range is:

  `range(start, end, step)`

- The default `step` (i.e increment) is 1
- We may also specify an increment:

```
# prints the numbers: 1,3,5,7,9
for i in range(1, 10, 2):
    print(i)
# prints the numbers: 2,4,6,8
for i in range(2, 10, 2):
    print(i)
# prints the numbers 5 to 1
for i in range(5,0, -1):
    print(i)
```

Using `range()`

- It is only required that the `end` argument be provided for the `range()` function.
- If the `start` argument is not provided, it is set as its default value of 0.

```
for i in range(4):
    print(i)
```

The above prints the numbers: `0,1,2,3` (remember, `end` is *not* inclusive)

the for and while loop

The `for` loop is like a short-hand for the `while` loop:

- ```
  i=0
  while i < 10:
      print(i)
      i += 1
  ```
- ```
  for i in range(0, 10, 1):
      print(i)
  ```

Common problems – Infinite Loops

Infinite loops are caused by an incorrect loop condition or not updating values within the loop so that the loop condition will eventually be false.

- Example:

```
n = 1
while n <= 5:
    print(n)
```

  Here we forgot to increase n $\rightarrow$infinite loop.

N.B. to exit from an infinite loop while running Python in the console, press Ctrl + C (press the stop icon in Jupyter Notebook).

Common Problems – Off-by-one Error

The most common error is to be "off-by-one". This occurs when you stop the loop one iteration too early or too late.

- Example:

```
for i in range(0,10):
    print(i)
```

  This loop was supposed to print 0 to 10, but it does not.

Example 25

Question: How can we fix this code to print 0 to 10?

Question: `for` loop

Example 26

How many numbers are printed in this loop

```
for i in range(1,10):
    print(i)
```

A) 0
B) 9
C) 10
D) 11
E) error

Answer:

How many numbers are printed in this loop

```
for i in range(1,10):
    print(i)
```

A) 0

B) *9*

C) 10

D) 11

E) error

Question: `for` loop

Example 27

How many numbers are printed in this loop

```
for i in range(11,0):
    print(i)
```

A) 0
B) 9
C) 10
D) 11
E) error

Answer:

How many numbers are printed in this loop

```
for i in range(11,0):
    print(i)
```

A) *0*

B) 9

C) 10

D) 11

E) error

Try it: for loops

Example 28

Write a program that prints the numbers from 1 to 10 then 10 to 1.

Example 29

Write a program that prints the numbers from 1 to 100 that are divisible by 3 and 5.

Example 30

Write a program that asks the user for 5 numbers and prints the maximum, sum, and average of the numbers.

Python Collections

There are four collection data types in the Python:

Type	Description	Allows duplicates
List	a collection which is ordered and changeable	Yes
Tuple	a collection which is ordered and unchangeable	Yes
Set	a collection which is unordered and unindexed	No
Dictionary	a collection which is unordered, changeable and indexed	No

Sidenote: if we can change it's values we call it **mutable**, if we can't change it's values, its said to be **immutable**.

List Overview

A list is a collection of data items that are referenced by index.

- Lists in Python are similar to arrays in other programming languages

A list allows multiple data items to be referenced by one name and retrieved by index.

- Python list:

Retrieving Items from a list

Items are retrieved by index (starting from 0) using square brackets:

- ```
  data = [100, 200, 300, 'one', 'two', 600]
  print(data[0]) # 100
  print(data[4]) # 'two'
  print(data[6]) # error ? out of range
  print(data[len(data)-1]) # 600
  print(data[-1]) # 600
  print(data[2:4]) # [300, 'one']

  # Create an empty list:
  emptyList = []
  ```
  ```

Retrieving Items from a list: Python List Slicing

List slicing allows for using range notation to retrieve only certain elements in the list by index. Syntax:

```
list[start:end:step]
```

- Example:

```
data = list(range(1,11))
print(data) # [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(data[1:8:2]) # [2, 4, 6, 8]
print(data[1::3]) # [2, 5, 8]
```

Sidenote: we could also use the `slice()` operator which works in the same manner. Usage: `slice(start, end, step)`

- Example:

```
>>> data[slice(1,8,2)]
[2, 4, 6, 8]
```

List Operations

We can modify a single value by use of indices and the assignment operator =:

```
>>> data = [1,2,3,5]
>>> data[2] = 7
>>> print(data)
[1, 2, 7, 5]
```

We can also modify multiple values at a time in the following way:

```
>>> data[0:1] = ["one","two"]
>>> print(data)
['one', 'two', 2, 7, 5]
```

List Operations

- Notice that when we try to add a value to the end of the list, we get an error:

```
>>> data[5] = 10
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
```

- To add an item to the end of the list, use `append()`.
- `append()` adds a single item to the existing list.
- It doesn't return a new list; rather it *modifies* the original list.

```
>>> data = [1,2,3,5]
>>> data.append(1)
>>> print(data)
[1, 2, 3, 5, 1]
```

List Operations

We can also use append in a for loop:

```
i = [1, 2, 3, 5, 8, 13]
j = []
for l in i:
    j.append(l)
```

Notice how we can iterate over a list in a for loop!

`j = [1, 2, 3, 5, 8, 13]`

List Operations

`extend()` on the other hand extends the list by adding all items of a list (passed as an argument) to the end.

It is best to see how this differs from `append` by way of example:

```
>>> x = [1, 2, 3]
>>> x.append([4, 5])
>>> print (x)
[1, 2, 3, [4, 5]]
```

where `x[3]` returns `[4, 5]` and `x[4]` returns an error.

```
>>> x = [1, 2, 3]
>>> x.extend([4, 5])
>>> print (x)
[1, 2, 3, 4, 5]
```

where `x[3]` returns `4` and `x[4]` returns `5`.

List Operations

```
data = [1, 2, 3, 5]
lst = []
```

Operation	Syntax	Examples	Output
Add item	<list>.append(val)	data.append(1)	[1, 2, 3, 5, 1]
Insert item	<list>.insert(idx,val)	data.insert(3,4)	[1, 2, 3, 4, 5]
Remove item	<list>.remove(val)	data.remove(5)	[1, 2, 3]
Update item	list[idx]=val	lst[0]=10	[10]
Length of list	len(<list>)	len(data)	4
Slice of list	list[x:y]	data[0:3]	[1, 2, 3]
Find index	<list>.index(val)	data.index(5)	3
Sort list	<list>.sort()	data.sort()	[1, 2, 3, 5]
Add	lst = []	lst.append(1)	[1]

To sort in reverse order, use `data.sort(reverse=True)`. See more here

List details

It was mentioned already but its worth repeating...

For loops that are used to iterate though items in a list:

```
data = [5,9,-2,9]
for v in data:
   print(v)

 #### output:
# 5
# 9
# -2
# 9
```

List details

Note that this is not restricted to numbers:

```
data = ["apples", "bananas","oranges"]
for v in data:
    print(v)
```

prints `apples`, `bananas`, `oranges` (each on a separate line).

We could even iterate through characters in a string:

```
for v in "bananas":
    print(v)
```

prints `b`, `a`, `n`, `a`, `n`, `a`, `s` (each letter on a separate line)

List details

If we want to iterate through both index and value, we could use the `enumerate()` function.

```
data = ['apple', 'banana', 'grapes', 'pear']
for c, value in enumerate(data):
    print(c, value)

# Output:
# 0 apple
# 1 banana
# 2 grapes
# 3 pear
```

List details

If we want our index to start at 1 rather than 0, we could specify that as the second argument: `enumerate()` function.

```
data = ['apple', 'banana', 'grapes', 'pear']
for c, value in enumerate(data, 1):
    print(c, value)

# Output:
# 1 apple
# 2 banana
# 3 grapes
# 4 pear
```

Advanced: Python List Comprehensions

List comprehensions build a list using values that satisfy a criteria.

- Example:
  `evenNums100 = [n for n in range(101) if n%2==0]`
- Equivalent to:

```
evenNums100 = []
for n in range(101):
    if n%2==0:
        evenNums100.append(n)
```

```
# another example:
>>> squares = [x**2 for x in range(10)]
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Example 31

At what index is item with value 3?

```
data = [1, 2, 3, 4, 5]
data.remove(3)
data.insert(1, 3)
data.append(2)
data.sort()
data = data[1:4]
```

A) 0 **B)** 1 **C)** 2 **D)** 3 **E)** not there

Answer:

At what index is item with value 3?

```
data = [1, 2, 3, 4, 5]
data.remove(3)
data.insert(1, 3)
data.append(2)
data.sort()
data = data[1:4]
```

A) 0 B) 1 C) *2* D) 3 E) not there

```
[2,2,3]
```

Try it: Lists

Example 32 (Question 1)

Write a program that puts the numbers from 1 to 10 in a list then prints them by traversing the list.

Example 33 (Question 2)

Write a program that will multiply all elements in a list by 2. Bonus: try doing this using the `enumerate()` function.

Example 34 (Question 3)

Write a program that reads in a sentence from the user and splits the sentence into words using split(). Print only the words that are more than 3 characters long. At the end print the total number of words.

Lists

In the previous example we use the `list()` function to create our list instead of the square brackets.

This constructs a list using the single input. This could be

- a sequence (eg. string, tuples) or
- a collection (set, dictionary) or
- a iterator object (like the objects iterated over in our `for` loops)

If no parameters are passed, it creates an empty list.

Tuples

- A tuple is a collection which is ordered and unchangeable. To create tuples we use round brackets ().

```
>>> thistuple = ("apple", "banana", "cherry")
>>> print(thistuple)
('apple', 'banana', 'cherry')
```

- Single elements in a tuple are referenced the same way as lists:

```
>>> thistuple[1]
'banana'
>>> thistuple[1:3]
('banana', 'cherry')
```

Tuples

- The empty tuple is written as an open and closed parentheses

```
>>> emptytup = ()
>>> print(emptytup)
()
```

- To write a tuple containing a single value you have to include a comma, even though there is only one value

```
>>> tup1 = (50,)
>>> print(tup1)
(50,)
>>> thistuple[2:3]
('cherry',)
```

Tuples - packing and unpacking

- We could also “pack" them without using parentheses:

```
>>> anothertuple = 1,2,3
>>> print(anothertuple)
(1, 2, 3)
```

- We can “unpack" the contents of a tuple into individual variables:

```
>>> a,b,c=anothertuple
>>> print(a)
1
>>> print(b)
2
>>> print(c)
3
```

Tuples

Unlike list, once a tuple is created, values can not be changed.

```
>>> thistuple = ("apple", "banana", "cherry")
>>> thistuple[1] = "pineapple"
TypeError: 'tuple' object does not support item assignment
```

Notice that tuples are also iterable, meaning we can traverse through all the values. eg,

```
for i in thistuple:
   print(i)
```

The above prints:

```
apple
banana
cherry
```

Python Sets

- A set—like a mathematical set—is a collection which is unordered and unindexed.
- Sets are written with curly brackets {}.

```
>>> # N.B they do not carry duplicates
>>> thisset = {"apple",  "cherry", "banana", "apple"}
>>> thisset
{'banana', 'cherry', 'apple'}
```

- Since sets are unordered, the items will appear in a random order and elements cannot be reference by index.
- Again we can iterate though each item using a `for` loop.
- Read more about these here and what types of methods can be performed on them.

Python Dictionary

A dictionary is a collection which is unordered, changeable and indexed. We create them with curly brackets and specify their keys and values.

```
thisdict = {
  "key1": "value1",
  "key2": "value2",
  "key3": "value3"
}
print(thisdict)
```

Output:

```
{'key1': 'value1', 'key2': 'value2', 'key3': 'value3'}
```

Python Dictionary

We can now reference elements by a given *name* (i.e key) rather than the standard integers index.

```
>>> thisdict['key1']
'value1'
```

```
dict = {1:'one', 2:'two', 3:'three'}
print(dict[1]) # one
print(dict['one']) # error - key not found
if 2 in dict:          # check if key exists
    print(dict[2]) # 'two'
dict[4] = 'four' # Add 4:'four'
del dict[1]  # Remove key 1
dict.keys()          # Returns keys
dict.values() # Returns values
```

Example 35

What value is printed?

```
data = {'one':1, 'two':2, 'three':3}
data['four'] = 4
sum = 0
for k in data.keys():
    if len(k) > 3:
        sum = sum + data[k]
print(sum)
```

A) 7 **B)** 0 **C)** 10 **D)** 6 **E)** error

Answer:

At what index is item with value 3?

```
data['four'] = 4
sum = 0
for k in data.keys():
    if len(k) > 3:
        sum = sum + data[k]
print(sum)
```

A) 7 B) 0 C) 10 D) 6 E) error

`[2,2,3]`

Try it: Dictionary

Example 36

Write a program that will use a dictionary to record the frequency of each letter in a sentence. Read a sentence from the user then print out the number of each letter.

- Code to create the dictionary of letters:

```
import string
counts = {}
for letter in string.ascii_uppercase:
    counts[letter] = 0
print(counts)
```

Summary of Python Collections

Lists can be altered and hold different types

```
lectures = [1,2,['excelI',excelII'],'CommandLine']
# can delete individual items
del lectures[3]
# reassign values
lectures[1] = 'Introduction'
type(lectures[1]) # <class 'int'>
type(lectures[2]) # <class 'list'>
type(lectures[3]) # <class 'str'>
```

Tuples are immutable (we can't change the values)

```
topics= (1,2,['excel1','excel2'],'CommandLine')
del topics[3]
# TypeError: 'tuple' object doesn't support item deletion
type(topics[2])
# <class 'list'>
```

Summary of Python Collections

Sets do not hold duplicate values and are unordered.

```
>>> myset={3,1,2,3,2}
>>> myset
{1, 2, 3}
>>> myset[1]
TypeError: 'set' object does not support indexing
```

Dictionaries holds key-value pairs (just like real life dictionaries hold word-meaning pairs). The continuation character (\) is used to split statements across multiple lines.

```
>>> wordoftheday = {\
 'persiflage':'light, bantering talk or writing' ,\
 'foment':'to instigate or foster'}
>>> wordoftheday['foment']
'to instigate or foster'
```

Functions and Procedures

A **procedure** is a sequence of program statements that have a specific task that they perform.

A **function** is a procedure that returns a value after it is executed.

Loosely speaking, functions are a special type of procedure for which we do not immediately know the result.

While there are many built in functions at our disposal in Python, we can also create own *user-defined functions*.

Defining and Calling Functions and Procedures

Creating a function involves writing the statements and providing a function declaration with:

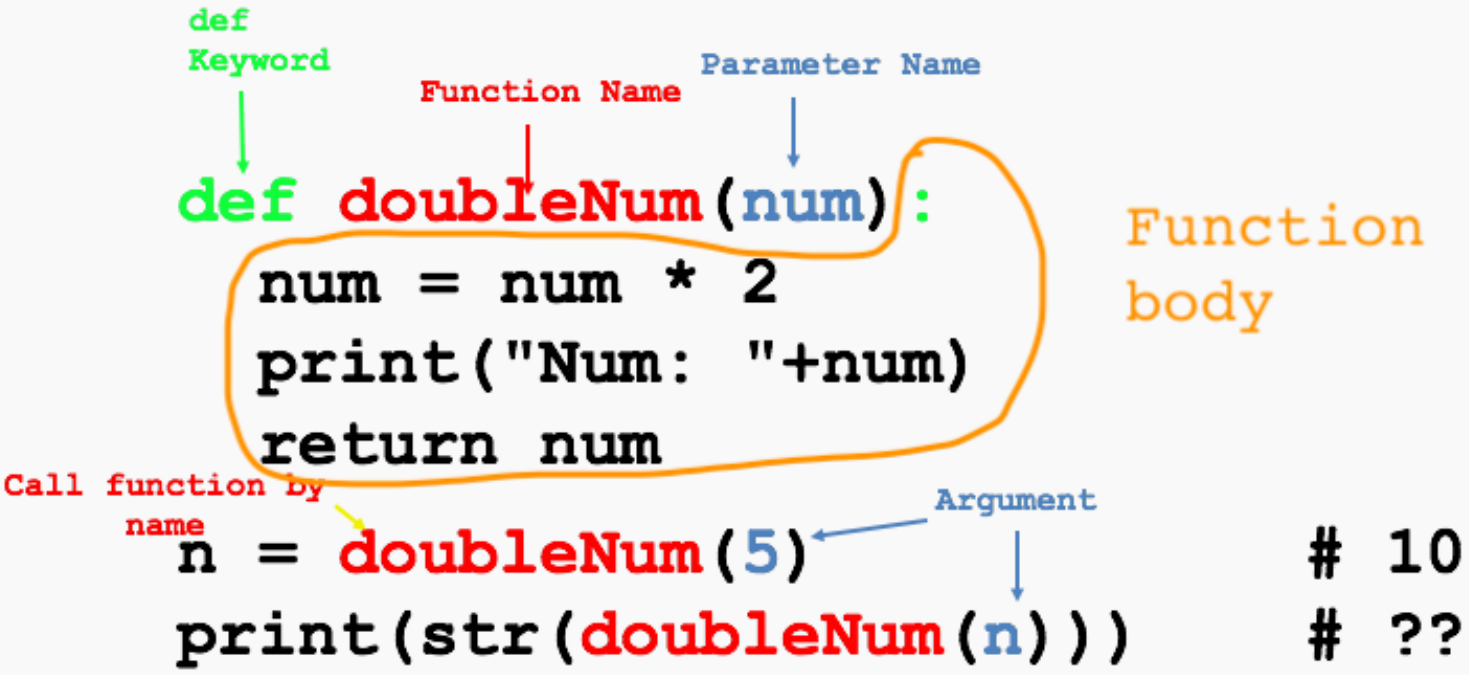
- a name (follows the same naming rules as variables)
- list of the inputs (called parameters)
- the output (return value) if any

Calling (or executing) a function involves:

- providing the name of the function
- providing the values for all arguments (inputs) if any
- providing space (variable name) to store the output (if any)

Defining and Calling a Function

Consider a function that returns a number doubled:



Defining and Calling a Function

- Function "blocks"[1] begin with the keyword `def` (short for define) followed by the function name.
- Regardless of whether or not the function has any parameters, we need to follow the function name with parentheses `()`
  - Inside the parentheses, separate as many parameters as you need by commas (no parameters should have the same name).
  - A function may have 0 parameter inputs.
- The code block within every function starts with a colon `:`
- The statements that form the body of the function starts from the next line of function definition and must be indented.

[1] A block is a piece of Python program text that is executed as a unit.

Functions and Procedures

See this procedure called `hi` that prints out `Hi!`

```
def hi():
    print("Hi!")
```

Calling this procedure twice (we know exactly what to expect each time):

```
>>> hi()
hi!
>>> hi()
hi!
```

See this function called `addf` which adds two numbers (or concatenates two strings)

```
def addf(x, y):
    return x + y
```

Calling the function with integers vs. strings:

```
>>> addf(2,5)
7
>>> addf("2","5")
'25'
```

Function Returns

- Function bodies can contain one or more `return` statement.
- The `return` statement exits a function and returns the value of the expression following the keyword.
- A function without an explicit `return` statement returns `None` (usually suppressed by the interpreter).
- Example:

```
def plus2(x):
    x + 2
```

Since we didn't specify a `return` statement, the calculation is not provided as output.

```
>>> plus2(3)
>>> nothing = plus2(3)
>>> print(nothing)
None
```

Function Returns

We will often save our return value(s) to an object defined within the function to be returned.

```
def testfun(x,y,z):
    out = x+y/z
    return out
```

Notice that the variables we define within our functions will not be defined outside of that function.

```
>>> testfun(3,8,4)
5.0
>>> out
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'out' is not defined
```

Function Returns

There can be multiple returns in a function but a function can only return exactly one *object*.

```
def gradeLetter(pgrade):
  if (pgrade >= 80):
     return "A"
  elif (pgrade >= 68):
    return "B"
  elif (pgrade >= 55):
    return "C"
  elif (pgrade >= 50):
    return "D"
  else:
    return "F"
```

```
>>> gradeLetter(81) # 'A'
>>> gradeLetter(45) # 'F'
```

Function Multiple Returns

If we want to return multiple values, we can return a list or a tuple, for example. Read here for some other ways.

- Example from here which returns the circumference and area of a circle with radius r (input):

```
def circleInfo(r):
    c = 2 * 3.14159 * r
    a = 3.14159 * r * r
    return (c, a)

print(circleInfo(10))
# (62.8318, 314.159)
```

To save it to individual variables we could unpack the tuple:

```
circumference, area = circleInfo(10)
print(circumference)  # 62.8318
print(area)   # 314.159
```

Function Multiple Returns

For this example we use a list to return two values instead of a tuple.

Both are acceptable but tuples are probably more common.

Some benefits of using tuples over lists are that:

- the results cannot be accidentally overwritten (tuples are immutable)
- and they can be easily unpacked into multiple variables

```
def grade(pgrade):
  if (pgrade >= 80):
    grade = "A"
  elif (pgrade >= 68):
    grade = "B"
  elif (pgrade >= 55):
    grade = "C"
  elif (pgrade >= 50):
    grade = "D"
  else:
    grade = "F"
  return [grade, pgrade]
```

```
>>> grade(81)
['A', 81]
```

Python Built-in Functions

Of course there are a multitude of built-in built-in functions, and methods in the standard Python library that makes life of a programmer easier. More examples here or here

- `max, min, abs:`

```
print(max(3, 5, 2)) # 5
print(min(3, 5, 2)) # 2
print(abs(-4)) # 4
```

- `type()` returns the argument data type:

```
print(type(42)) # <class 'int'>
print(type(4.2)) # <class 'float'>
print(type('spam')) # <class 'str'>
```

Python `math` module

It is important to keep in mind that there are many useful functions available to us in modules. For example, some functions available in the `math` module that can help us with mathematical calculations.

```
# Math
import math
print(math.sqrt(25))

# Import only a function
from math import sqrt
print(sqrt(25))

# Print all math functions
print(dir(math))
```

Python Random Numbers

Another useful model is the `random` module. We can for instance import the `randint` function to generate random numbers.

Usage: `randint(a, b)` returns a random integer `N` such that $a \leq N \leq b$.

```
from random import randint
coin = randint(0, 1)          # 0 or 1
die = randint(1, 6)          # 1 to 6
print(coin)
print(die)
```

Advanced: Python Functions

Python supports functional programming allowing functions to be passed like variables to other functions.

- Lambda functions are helpful in this context as they are small anonymous functions.
- In other words they do not require the `def` keyword or a function name.
- Example: take a function as the first argument `func` with it's input as the second argument `val`. Return the output `func(val)`.

```
def doFunc(func, val):
    return func(val)

print(doFunc(doubleNum, 10)) # 20
print(doFunc(lambda x: x * 3, 5)) # 15
```

Example 37

What is the value printed:

```
def triple(num):
    return num * 3

n = 5
print(triple(n)+triple(2))
```

A) 0 **B)** 6 **C)** 15 **D)** 21 **E)** error

Answer:

What is the value printed:

```
def triple(num):
    return num * 3

n = 5
print(triple(n)+triple(2))
```

A) 0 B) 6 C) 15 D) *21* E) error

Practice Questions: Functions

Example 38

1) Write a function that returns the largest of two numbers.

2) Write a function that prints the numbers from 1 to N where N is its input parameter.

Call your functions several times to test that they work.

Conclusion

Python is a general, high-level programming language designed for code readability and simplicity.

Programming concepts covered:

- variables, assignment, expressions, strings, string functions
- making decisions with conditions and if/elif/else
- repeating statements (loops) using for and while loops
- reading input with input() and printing with print()
- data structures including lists and dictionaries
- creating and calling functions, using built-in functions (math, random)

Python is a powerful tool for data analysis and automation.

Objectives

- Explain what is Python and note the difference between Python 2 and 3
- Define: algorithm, program, language, programming
- Follow Python basic syntax rules including indentation
- Define and use variables and assignment
- Apply Python variable naming rules
- Perform math expressions and understand operator precedence
- Use strings, character indexing, string functions
- String functions: split, substr, concatenation
- Use Python datetime and clock functions
- Read input from standard input (keyboard)

Objective (cont'd)

- Create comparisons and use them for decisions with if
- Combine conditions with and, or, not
- Use if/elif/else syntax
- Looping with for and while
- Create and use lists and list functions
- Advanced: list comprehensions, list slicing
- Create and use dictionaries
- Create and use Python functions
- Use built-in functions in math library
- Create random numbers
- Advanced: passing functions, lambda functions