Functions – Return Values

- Returns None
  - Output not reusable

```
def fxn_avg (a, b):
    sum = a + b
    avg = sum/2
    print (avg)

y = fxn_add (10, 10)
print (y) # None
type(y)
# <class 'NoneType'>
```

- Returns a value
  - Output reusable

```
def fxn_avg (a, b):
    sum = a + b
    avg = sum/2
    return avg

y = fxn_add (10, 10)
print (y) # 10
type(y)
# <class 'float'>
```

Functions – Return Values

- Only ONE value returned
  - Comma seperated values implicitly converted into tuple

```
def fxn_avg (a, b):
   sum = a + b
   avg = sum/2
   return sum, avg

y = fxn_add (10, 10)
print (y) # (20, 10.0)
type(y) # <class 'tuple'>
```

Functions – Duplication

- Adding arbitrary numbers

```
def fxn_add (*args):
    sum = 0
    for i in args:
        sum += i
    return sum

fxn_add (1,2,3)
```

- Average of arbitrary numbers

```
def fxn_avg (*args):
    sum = 0
    count = len(args)
    for i in args:
        sum += i
    avg = sum/count
    return avg

fxn_avg (1, 2, 3)
```

Functions – Reuse& Deduplication

```
def fxn_add (*args):
    sum = 0
    for i in args:
        sum += i
    return sum

fxn_add (...)

```

```
def fxn_avg (*args):
    sum = fxn_add(*args)
    count = len(args)
    avg = sum/count
    return avg

fxn_add ()

```

Functions – Reuse& Deduplication

- Without reuse

```
def fxn_avg (*args):
  sum = 0
  count = len(args)
  for i in args:
     sum += i
  avg = sum/count
  return sum

fxn_avg (...)
```

- With reuse

```
def fxn_avg (*args):
  sum = fxn_add(*args)
  count = len(args)
  avg = sum/count
  return avg

fxn_add ()
```

Functions – More on Reuse

- Module files

```
# csc1017f.py

def fxn_1 ():
   return “fxn_1”

def fxn_2 ():
   return “fxn_2”

def fxn_3 ():
   return “fxn_3”
```

- Module reuse

```
import csc1017f

csc1017f.fxn_1()
csc1017f.fxn_2()

from csc1017f import fxn_3

fxn_2()

fxn_add ()
```

STOP N – Examples with Builtins

- Would this work? What is the result? What is the return type?
  - y = print(“hello”)
  - y = len(“four”)
  - y = input(“Enter value:”)

UCT Department of Computer Science
Computer Science 1017F

Testing

Lighton Phiri <lphiri@cs.uct.ac.za>
April 2015

Introduction

- What is an error?
  - When your program does not behave as intended or expected.
- What is a bug?
  - "...a bug crept into my program ..."
- Debugging
  - the art of removing bugs

Types of Errors – Syntax Errors

- Syntax Errors
  - Failure to conform to Pythonic syntactic rules
  - Improper use of Python language – usually Syntax Errors.
    - 12var = 2

```
File "<stdin>", line 1
12var = 2
    ^
SyntaxError: invalid syntax
```

- Fixing Syntax Errors
  - Easiest type of error to fix
    - Conform! Follow the rules!

Types of Errors – Runtime Errors

- Runtime Errors
  - Program is syntactically correct but error detected after execution.
    - `a = 1/0 (ZeroDivisionError: division by zero)`
    - `a = b + 10`
    - `3 "four"[100] (IndexError: string index out of range)`
    - `I/O operations`
- Fixing Runtime Errors
  - Exception handling.
  - Normally able to anticipate potential program flaws
    - Examples above illustrate this

Types of Errors – Runtime Errors

- Exception handling involved

```
def fxn_divide (a, b):
  result = 0
  try:
     result = a / b
  except ZeroDivisionError:
     print (“Division by ZERO error!”)
  return result

fxn_divide(1, 0)
```

Types of Errors – Logical Errors

- Logical Errors
  - Program parsed by by interpreter and runs successfully, but produces undesirable results
    - `sum = 1 - 1`
    - `Conditionals—logical operators—and, or, not`

- Fixing Logical Errors
  - Programmer responsibility to ensure code functions as required before releasing it
    - Debugging
    - Testing

STOP 1: Identifying Errors

```
def add_x (a, b):
    return a + b
    print ("Output") # Logical Error

add_x("1", "2") # Logical Error
add_x(+) # Syntax Error
add_x() # Runtime Error
```

- Identify errors in code snippet—what types are they?
- An individual recently won a $5k Google bounty for figuring out that YouTube videos could easily be deleted. What error was identified?

Debugging

- **Debugging** is the process of finding **errors** or **bugs** in the code.
- Debugging techniques
  - Application stacktraces—Android app, Web browsers, OSes
- A **debugger** is a tool for executing an application where the programmer can carefully control execution and inspect data.
- Features include:
  - step through code one instruction at a time
  - viewing variables
  - insert and remove breakpoints to pause execution

Basic Debugging Techniques

- Commenting out code thought to be root cause of problem

```
### result = a/b # possible division by 0
if b > 0
  result = a/b
```

- Tracing
  - Builtin output function

```
print (a) # check output value
if a > 0
  result = a/b
```

Debugging with Wing

- Hands on Laboratory Exercise
- Set breakpoints in the code to halt execution path
- Initiate execution in debug mode, start debugging
- Facilitate user input/output in I/O Debug window
- Step through code and view—in real-time—variable change in stack data window
- Stop debugging when error is located

STOP 2: More Debugging Techniques

- Logging—e.g. MS Event& Error logs
  - Logging package

```
import logging
logging.basicConfig(filename=”y.log”)
except :
  logging.error(“Error”) # critical|warn|info|debug
```

  - print() statement

```
except:
  print ("Error", file=open(“y.log”, mode="w"))
```

- Auxiliary Tools
  - Python modules: pdb, pylint, pep8

Testing Levels

- Unit testing
  - Individual components of code
    - Functions; classes
    - e.g. fxn_palindrome(); fxn_palindromic_prime()
- Integration testing
  - Interaction between components' interfaces
  - e.g. fxn_palindrome() and fxn_palindromic_prime()
- System testing
  - End-to-end testing of integrated system
    - e.g. If we built a system for computing RSA taxation...

Testing Methods

- Programs should be thoroughly tested for all potential input/output values to ensure code behaves as desired
- In most cases it is not feasible to test all possible input values
  - Palindrome laboratory exercise
  - add_n(var_1, var_2)
- Test inputs should be appropriately chosen to ensure code is thoroughly tested
  - Test cases

Equivalence Classes& Boundary Values

- Equivalence classes are used to get test values from categories with similar behaviour
  - Category values from each category
  - Erroneous values
- Boundary values are used to test extreme values at boundaries
  - On boundary values
  - Below boundary values
  - Above boundary values

Path Testing& Statement Coverage

- Path testing involves testing every path of execution of the program at least once

```
if a > 0:
  print ("Error")
```

  - `Path A: a=10; Path B: a=-10 # Test both paths`

- Statement coverage involves testing every statement of the program at least once

```
if a > 0:
  print ("Error 1")
if b < 0:
  Print ("Error 2")
```

  - `e.g. (a, b) = (10, 10); (-5, -5)`

Glass and Black Box Testing

- Black Box testing involves developing test cases only based on the problem specification and NOT the actual code

- Glass Box testing involves developing test cases based on the code

STOP 3: Classification

- Are the following techniques Black Box testing or Glass Box testing?
  - Equivalence classes/boundary values
  - Path coverage
  - Statement coverage

Test Cases

- Each input case needs a corresponding manually calculated expected output specified

- When program is tested the actual output generated by code is compared to expected output

STOP 4: Test Case Example

```
# add positive numbers
def add_pos (a, b):
    if a > 0 and b > 0:
        return
```

Step 1: manual generation

Input | Expected output (manual)

Step 2: comparison

Input | Expected | Generated